Universidad Nacional de La Plata

Facultad de Informática

Implementación de librería para interpretación y análisis de paquetes de red

Matias Fontanini

Director:
Luis Marrone

Co-directora:
Paula Venosa

Índice

1 Capítulo 1: Introducción

1.1 Motivación

El desarrollo de herramientas que realicen análisis o inyección de trafico en redes ha existido durante décadas. Para cada una de ellas, sus autores debieron utilizar algún mecanismo que les permitiera interpretar el trafico capturado de los dispositivos para luego poder procesarlo correctamente.

Debido a que no existe una librería que permita realizar esa interpretación de una forma portable y sin tener un impacto significativo en la *performance* de la aplicación desarrollada, éstos suelen hacer una implementación *ad-hoc*.

Esto puede ser muy tedioso y propenso a errores dado que se debe tener en cuenta múltiples aspectos:

- Las estructuras y mecanismos internos utilizadas por cada protocolo al que se quiera dar soporte.
- El *endian* de cada protocolo, que puede ser diferente del de la arquitectura sobre la que se corra la aplicación, lo que además puede generar problemas de portabilidad.

Además se termina generando una replicación de código muy grande: cada desarrollador reimplementa el mecanismo de interpretación de paquetes, diseñándolo y manteniéndolo atado a su aplicación.

Si bien existen herramientas muy potentes, como *Wireshark*[1], que permiten capturar paquetes de red y analizarlos en tiempo real, éstas no están preparadas para inspeccionar grandes volúmenes de tráfico. Además, no siempre permiten hacer el análisis que se desea, por lo que que es necesario implementar una aplicación que lo haga.

Durante el 2011, se intentó implementar una aplicación que capturara tráfico y realizara diferentes acciones dependiendo de los paquetes de red que se vieran. Esta estaba siendo desarrollada en el lenguaje *Python*[2] utilizando *scapy*[3], una herramienta y librería de captura de paquetes. Al probar algunos módulos de la aplicación en una red en la que se veía un volumen moderado de tráfico, se notó que había pérdidas de paquetes. Si bien se probaron diferentes mecanismos para intentar acelerar la captura, nada parecía dar resultado. El problema no era la aplicación, sino la librería de captura de paquetes. Ésta era muy lenta para analizar e interpretar paquetes, por lo que no se daba a basto y éstos se terminaban perdiendo. Esto fue el motivo principal por el que se comenzó el desarrollo de *libtins*, una librería de código abierto que permitiera el envío y captura de paquetes de red.

1.2 Objetivos

Los objetivos que se le plantearon a la librería desarrollada fueron los siguientes:

- Capturar e interpretar paquetes de red. Esto es, no sólo leerlos de una interfaz de red, sino también separarlos en las diferentes capas que lo componen para facilitar su análisis. Esto implicaría permitir el acceso al contenido de cada campo presente en la estructura del paquete.

- Alterar y enviar paquetes de red. Muchas aplicaciones necesitan hacer envío de paquetes a bajo nivel para interactuar de alguna manera con otras máquinas o servicios, pero no tienen acceso a un mecanismo simple y eficiente para hacerlo. Como consecuencia, agregar esta característica permitiría que la librería sea útil no sólo para quienes quieran hacer análisis pasivo de tráfico, sino también para quienes deseen llevar a cabo acciones activas en la red.

El objetivo de la librería sería poder utilizarla para una gran variedad de tareas que impliquen análisis de tráfico o envío de paquetes a bajo nivel. Entre otras, podrían listarse:

- Detección de actividades maliciosas. Se podrían aplicar filtros que permitan detectar tráfico que pareciera tener fines maliciosos, como explotación de vulnerabilidades ó detección de ataques de fuerza bruta contra los diferentes servicios de una red para luego generar alertas o incluso bloqueos automatizados.

- Monitoreo de redes. Esto podría ser utilizado para aplicar control de acceso sobre los usuarios de una red o simplemente para conocer a qué servidores realizan conexiones.

- Detección de anomalías en la red. Se podrían buscar situaciones que parezcan ser producto de problemas en la red, como pérdidas y reenvío de paquetes, incremento del *round-trip time* (RTT) entre los dos extremos de una conexión *TCP*, etc. Esto permitiría luego enfocar el diagnóstico de los problemas adonde corresponda.

1.3 Comienzo del desarrollo

Este desarrollo se inició en Agosto de 2011 junto a Santiago Alessandri, un compañero de la *Facultad de Informática* de la *Universidad Nacional de*

La Plata. A partir de Marzo de 2012 y hasta la actualidad, el desarrollo y mantenimiento de la librería fue continuado por quien escribe, además de algunos parches aportados por usuarios de la misma. En el momento en el que el desarrollo fue continuado en forma individual, la librería estaba en un estado inestable, recién se estaba iniciando la etapa de *testing* y no funcionaba en ningún sistema operativo más que *GNU/Linux*.

Durante este tiempo, se trabajó bastante para lograr un proyecto estable, fácil de usar y bien documentado. Esto incluyó el desarrollo de un sitio *web* para que cualquier usuario pueda acceder a tutoriales y la documentación completa de la librería. Este se encuentra actualmente alojado en la siguiente *URL*:

```
http://libtins.github.io/
```

Por otro lado, el código fuente también se encuentra alojado en la plataforma *github*. Para acceder a él se puede entrar a la siguiente *URL*, de donde se lo puede descargar o mirarlo desde un navegador.

```
https://github.com/mfontanini/libtins
```

Para que cualquier usuario de la librería se pueda sacar dudas sobre como utilizar la librería o desee reportar errores encontrados, se creó un foro en el que pueden hacerlo. Este está alojado en la siguiente *URL*:

```
https://groups.google.com/forum/#!forum/libtins
```

1.4 Estructura de la Tesina

Esta Tesina se organiza de la siguiente manera:

- *Capítulo 1*: se introduce el tema sobre el cuál se desarrolló, indicando motivaciones y objetivos propuestos.
- *Capítulo 2*: introduce conceptos de redes que son necesarios para entender la implementación desarrollada.
- *Capítulo 3*: se muestran los diferentes criterios de diseño que se tuvo en cuenta al llevar a cabo este proyecto.

- *Capítulo 4*: éste contiene en detalle las características más relevantes de la librería indicando cuáles fueron los criterios que llevaron a implementarlas de esa manera.

- *Capítulo 5*: esta sección contiene el resultado de las pruebas que se hicieron para comparar la velocidad de *libtins* contra otras librerías que realizaran las mismas tareas.

- *Capítulo 6*: en ésta se incluyen algunos ejemplos que muestran la utilidad y simplicidad de la librería.

- *Capítulo 7*: esta sección contiene conclusiones y propuestas de modificaciones que se podrían realizar en el futuro sobre el trabajo.

2 Capítulo 2: Conceptos de redes

Al desarrollar aplicaciones que envíen datos a través de *sockets*, uno se abstrae completamente de los protocolos que se utilizan para permitir que lleguen a destino. Es el *kernel* de nuestro sistema operativo el que se encarga de proveernos esta interfaz transparente para realizar envío y recepción de datos. Sin embargo, sin que lo notemos éste los encapsula con diferentes estructuras que permiten que sean entregados al receptor, incluso si deben pasar por múltiples dispositivos intermedios.

Éstas estructuras tienen varios objetivos, como por ejemplo:

- Sirven para identificar quien emite y quien debería recibir los datos que se envían.
- A menor escala, sirve para identificar dispositivos emisores y receptores intermedios que cooperan para que los datos lleguen al destino final.
- A veces por anomalías en la red, los datos pueden corromperse. Por eso también incluyen mecanismos de chequeo de errores que permiten garantizar que los datos que se reciben son los mismos que se envían.
- Algunas proveen mecanismos para detectar si los datos llegan a destino o se deben reenviar porque se perdieron como consecuencia de alguna anomalía en la red.

A estas estructuras que utiliza el *kernel* para encapsular los datos que se envían se las denomina *PDU*(unidad de datos de protocolo). Cada una de éstas contiene los datos correspondientes a algún protocolo que se utiliza en la red. Si bien todas tienen objetivos y formatos diferentes, se las puede agrupar dependiendo de su función principal. Una de las formas de catalogarlas es utilizando el modelo *TCP/IP*[6].

2.1 Modelo *TCP/IP*

El modelo conceptual *TCP/IP* identifica y caracteriza las funciones de un sistema de comunicaciones particionándolo en diferentes capas. Cuando una aplicación envía datos a través de la red, estos son encapsulados con estructuras correspondientes a cada una de ellas, y así logrando que los datos lleguen al receptor.

Cuando se envían datos desde una aplicación, estos se fraccionan en fragmentos pequeños y cada uno de estos se encapsula con esas capas. Los datos junto con las estructuras que se le anexan para proveer las garantías

que se mencionaron anteriormente se denomina paquete. Un paquete es la unidad mínima de transferencia de datos en la red. Es decir, si se quiere enviar cualquier tipo de datos, estos tienen que encapsularse en un paquete para poder hacerlo.

Las diferentes capas que componen a un paquete se definen de la siguiente manera:

2.1.1 Capa 1 (física)

Esta es la más cercana al *hardware*, y define las especificaciones eléctricas y físicas de una conexión de datos. Asimismo, define la relación entre un dispositivo y un medio de transmisión físico, como puede ser cobre o fibra óptica, y los voltajes, tiempo de señales, especificaciones de cable y otras características que se deben utilizar para transferir datos.

Los protocolos utilizados en la capa física no serán analizados, dado que estos, si bien son fundamentales para la transmisión confiable de datos entre dos nodos, son manejados internamente por los dispositivos de red y no son manipulables desde $software$.

2.1.2 Capa 2 (capa de enlace)

Esta capa provee un medio confiable para transferir datos entre 2 nodos (computadoras, *switches*, *routers*, etc) conectados directamente entre ellos. En ésta se detectan e intentan solucionar errores posiblemente encontrados en la capa 1. Los protocolos utilizados en esta capa pueden ser *Ethernet*[7], *PPP (point-to-point-protocol*[8] e *IEEE 802.11*[9], entre otros.

Al capturar tráfico de un dispositivo de red, esta es la capa más baja que se encontrará en cada paquete y siempre estará presente. Si bien algunos de los protocolos de esta capa tienen estructuras difíciles de analizar, como *RadioTap*[10], es importante dar soporte a la mayor cantidad posible de ellos. De otra manera, si un dispositivo de red está configurado para utilizar algún protocolo de capa de enlace que no esté implementado en la librería, no se podrá analizar ninguno de los paquetes que se capturen a través de él.

2.1.3 Capa 3 (capa de red)

Esta capa tiene el objetivo de permitir el correcto enrutamiento de paquetes entre nodos que se encuentren en una misma o en diferentes redes. Los protocolos utilizados en esta capa permiten identificar al emisor y receptor de cada paquete, y así permitir que cualquier dispositivo que se encuentre en

la ruta de un paquete pueda enviarlo hacia adonde corresponda. Ejemplos de protocolos de esta capa son *IPv4*[11] e *IPv6*[12].

Los protocolos de esta categoría, en general, no proveen ningún tipo de garantía de que un paquete llegue a destino. Por otro lado, éstos suelen utilizar mecanismos para detectar alteraciones en el contenido de los paquetes generados por errores en la red. Esto se logra aplicando una función numérica sobre el contenido de esta capa y las superiores, y luego guardando el valor resultante dentro de la cabecera del protocolo. El receptor luego aplicará la misma función o comprobará que el valor sea el mismo. La correcta implementación de esta función es fundamental para poder generar y enviar paquetes. De otra manera, estos serán descartados por el destinatario.

Los protocolos de estas capas también incluyen opciones (como por ejemplo *IPv4* e *ICMPv6*) o cabeceras de extensión (en el caso de *IPv6*). Estas estructuras utilizan el formato *TLV*, el cual es descripto en la sección 2.2.1.

2.1.4 Capa 4 (capa de transporte)

Esta capa permite enviar datos de forma confiable entre un nodo emisor y uno receptor. Además, provee diferentes servicios como soporte de flujos de datos orientados a conexión, confiabilidad, control de flujos y multiplexación.

Los protocolos más comúnmente vistos de esta capa son *TCP (transmission control protocol)*[13] y *UDP (user datagram protocol)*[14], aunque también existen otros como *SCTP(stream control transmission protocol)*[15].

Al igual que los protocolos de capa de red, los de capa de transporte tienen un mecanismo similar de chequeo de errores. Algunos, como *TCP* y *UDP*, utilizan campos del protocolo de capa 2 que se utilice en el paquete como parte de los datos que alimentan la función descripta en la sección 2.1.3. Esto implica que al momento de serializar un paquete, es necesario que las capas superiores puedan acceder a las inferiores.

Algunos protocolos de esta capa, como *TCP*, utilizan opciones en formato *TLV* al igual que los de capa de red.

2.1.5 Capa 5 (capa de aplicación)

La capa de aplicación abstrae los protocolos utilizados por aplicaciones para comunicarse entre ellas. Todos los datos que envíe cualquier proceso que utilizamos diariamente (navegadores web, clientes de correo, etc) se encontrará en ésta.

Los protocolos de capa de aplicación, en su mayoría, no son de nuestro interés desde el punto de vista de la librería que se desarrolló. Esto se debe

a que:

- Van más allá de un simple modelado de paquetes: los datos que se envían en ellos suelen estar divididos en múltiples paquetes (por ejemplo porque se utiliza *TCP*) como pueden ser transacciones *HTTP*[16]. Como consecuencia, y a diferencia de los protocolos de capas inferiores, no siempre se puede analizar un único paquete y extraer datos de esta capa, sino que generalmente hay que rearmar flujos y recién ahí analizar el contenido.

- Son demasiados: existen miles de protocolos de capa de aplicación, por lo que llegar a modelar al menos una parte significativa implicaría muchísimo trabajo, que nuevamente no se justifica por el punto anterior.

A pesar de los puntos mencionados, hay ciertos protocolos de capa de aplicación que son fundamentales en cualquier red de computadoras. Como consecuencia, cualquier librería que permita analizar y manipular paquetes debe darles soporte. Ejemplos de estos protocolos pueden ser *DNS (Domain Name System*[17] y *DHCP (Dynamic Host Configuration Protocol)*[18].

2.2 Estructuras de los protocolos

Las estructuras de los protocolos utilizados en las capas anteriores a la de aplicación suelen tener características en común:

- Son compactos: se quiere agregar un *overhead* chico a cada paquete, por lo que no es conveniente utilizar estructuras extensas o que desaprovechen espacio. En general, todas suelen tener al menos una estructura base con un tamaño fijo, compuesto por campos de longitudes que normalmente pueden variar entre 1 y 128 *bits*. Algunas, como las de los protocolos *IP* o *TCP*, agregan una sección de opciones, las cuales también están definidas de forma compacta.

- Fácil análisis: los paquetes viajan por diferentes dispositivos hasta llegar a destino. Cada uno de estos tiene que tomar decisiones de enrutamiento en base su contenido. Por ejemplo, un *router* puede decidir por qué interfaz enviarlo en base a su dirección *IP* destino, o un *firewall* puede decidir si descartarlo o no en función de sus atributos internos. Todos estos dispositivos (en principal los *routers* o *switches*) son pequeños y no tienen mucho poder de cómputo. Es por esto que

la estructura de los protocolos debe ser fácil de analizar y encontrar los datos que se están buscando.

2.2.1 Opciones en formato *TLV*

Como se mencionó en la sección 2.2, algunos protocolos suelen utilizar una sección de datos opcionales. Muchos de estos suelen utilizar el formato *TLV* (*type-length-value* o tipo-longitud-valor) para representarlas, por su simplicidad y forma compacta. El formato de esta estructura puede verse en la figura 1. Este está compuesto por:

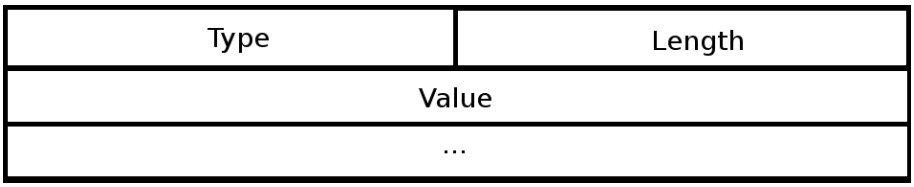

Figura 1: *Opción TLV*

- Campo *type*: este suele ser de uno o dos *bytes* e indica el tipo de la opción.
- Campo *length*: este indica la longitud del campo *value*. Comúnmente es de uno o dos *bytes*.
- Campo *value*: este contiene el valor de la opción.

Esta estructura es comúnmente utilizada para almacenar datos opcionales dado que como puede verse, ocupa poco espacio para cada opción y no tiene *overhead* alguno si no se utiliza ningún dato opcional.

2.2.2 *Endian* de las estructuras

Existen diferentes formatos, denominados *endian*[19], que indican la forma en la que se almacenan datos de una longitud mayor a un *byte* en una computadora. Cada protocolo almacena sus datos en un formato determinado, por lo que es necesario saber cuál es para poder interpretar sus datos correctamente

Si bien existen varios *endian* diferentes, los más utilizados son los siguientes:

- *Big endian*: consta de ubicar el *byte* más significativo de los datos que se quieren almacenar al comienzo y el resto a continuación, siguiendo el orden original. La figura 2 muestra un ejemplo de como se organiza el contenido de un registro en memoria.

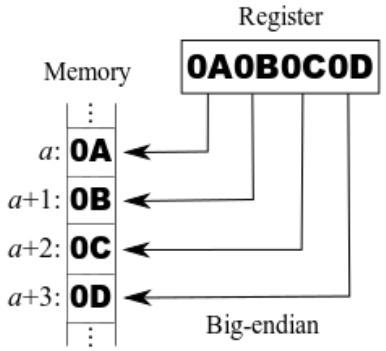

Figura 2: *Big endian*

- *Little endian*: esta es exactamente lo opuesto a *big endian*. El *byte* menos significativo se ubica al comienzo y a continuación el resto, nuevamente siguiendo el orden original, tal como muestra la figura 3.

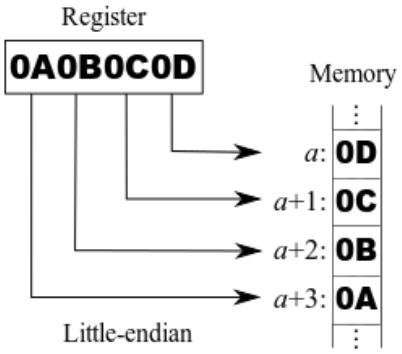

Figura 3: *Little endian*

El término *network order* hace referencia al *endian* que se utiliza en los protocolos de red. Entre otros, el *RFC 1700* indica que este debe ser *big endian*, por lo que todos los protocolos que se mencionaron en las secciones anteriores utilizan este formato. A pesar de esto, existen algunos protocolos de red que utilizan el formato *little endian* para almacenar sus datos, como

puede ser *RadioTap* o *IEEE 802.11*.

Las computadoras que usamos a diario suelen utilizar el formato *little endian* internamente. Por otro lado, otros dispositivos que utilizan arquitecturas *MIPS* o *ARM*, como *routers*, suelen ser *big endian*. Es por esto que hay que tener en cuenta esta posible diferencia entre el formato que utilice la maquina sobre la que se corran las aplicaciones que analicen paquetes y el de los protocolos en sí para evitar incompatibilidades y posibles interpretaciones erróneas del contenido de los paquetes.

2.2.3 Apilamiento de *PDU*s

Como se mencionó en las secciones anteriores, cada paquete está compuesto por diferentes *PDU*, cada uno correspondiente a una capa del modelo *TCP/IP*. La figura 4 muestra como estaría compuesto un paquete *TCP*.

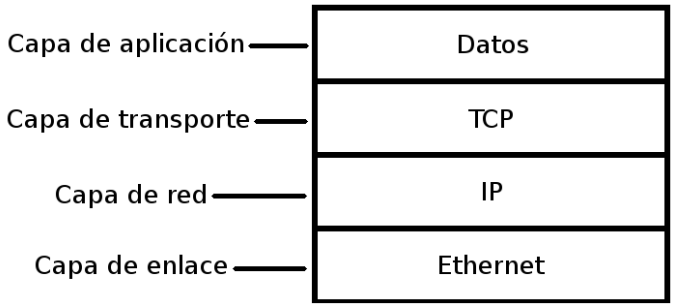

Figura 4: *PDUs* en un paquete *TCP*

Todos los *PDU*, excepto los pertenecientes a capa de transporte, incluyen algún campo que indica cual es el tipo que le sigue, lo que permite anidar múltiples capas independientes en un mismo paquete. Por ejemplo, el *RFC 1700* define los números que se utilizan dentro del protocolo *IP* para identificar al siguiente protocolo en la pila de *PDU* que conforman un paquete.

2.2.4 Chequeo de errores

Muchos protocolos implementan mecanismos de chequeo de errores para poder comprobar que un paquete no fue alterado por errores en la red una vez que éste llega a destino. Esto suele implementarse utilizando un campo dentro de la estructura del protocolo que contiene un valor numérico, el cual es el resultado de aplicar una función sobre el contenido del paquete. Una

vez que este termina en el receptor, se utiliza la misma función para verificar que el valor del campo sea correcto.

La función que se utiliza para obtener este valor suele implicar la suma de todos los *bytes* que contenga el paquete y luego la aplicación de alguna función de manipulación de *bits*.

2.3 Documentación de los protocolos

La mayor parte de los protocolos que se encuentran en cualquier red de computadoras pasan por varias etapas antes de que se llega a una documentación formal que indique su estructura y modo de uso. Las excepciones pueden ser protocolos privativos o de ciertas aplicaciones que no necesitan que ellos sean estandarizados.

Inicialmente, la documentación de los nuevos protocolos suele estar publicada en uno o más *RFC (Request For Comments)*[20]. En éstos se definen diferentes atributos y características de ellos, y son publicados de forma gratuita para que cualquier persona pueda leerlas y en caso de ser necesario, desarrollar sus propias implementaciones. Algunos *RFC* o conjuntos de ellos pueden terminar derivando en un estándar publicado por la *IETF (Internet Engineering Task Force)*[21].

También existen estándares publicados por otras organizaciones, como la *IEEE (Institute of Electrical and Electronics Engineers)*, la *W3C (World-Wide Web Consortium)* o la *ITU (International Telecommunication Union)*. Los de la primera apuntando a protocolos más cercanos al *hardware*, como puede ser *IEEE 802.11*, la segunda estando más relacionados a la *Web*, como *HTML (HyperText Markup Language)*[22] o *XML (eXtensive Markup Language)*[23] y la última apuntando a estándares de vinculados a las telecomunicaciones.

La lectura y el análisis de los diferentes *RFC* y estándars es fundamental para la correcta implementación de las estructuras de los protocolos a los que se quiere dar soporte.

3 Capítulo 3: Criterios de diseño

Antes de comenzar el diseño de la librería, se analizaron diferentes cuestiones de la implementación que era era necesario definir para poder comenzar su implementación. Estas no sólo afectarían el resultado final del proyecto, sino también su desarrollo.

3.1 Lenguaje de programación

Para elegir un lenguaje de programación adecuado para el desarrollo de la librería, se tuvieron en cuenta diferentes características:

- Portabilidad: era necesario elegir un lenguaje que pudiese funcionar en diferentes sistemas operativos, como *GNU/Linux*, *Windows* o *BSD*.
- Velocidad: además, debía ser un lenguaje que permita que la librería utilice eficientemente los recursos de la máquina en la que se corra, para que como consecuencia permita procesar una gran cantidad de paquetes por segundo.
- Manejo de memoria: dado que muchos protocolos utilizan campos de un tamaño no entero de *bytes* (es decir, de cualquier cantidad de *bits* y no necesariamente múltiplo de 8), era deseable que el lenguaje nos permita definir y manipular campos de este tipo.

Por estos motivos se eligió el lenguaje *C++*, el cual posee las siguiente características:

- Es multiparadigma: posee características de programación imperativa, orientada a objetos y genérica.
- Es un lenguaje compilado: su código fuente es traducido a código maquina y es ejecutado directamente por el *CPU*.
- Permite hacer manejo de memoria a bajo nivel. Esto es especialmente útil para poder definir y manipular las estructuras internas de cada protocolo.
- Es un lenguaje multiplataforma: los diferentes compiladores que existen han sido portados a múltiples sistemas operativos y arquitecturas, lo que permite tomar el código fuente, compilarlo en una plataforma determinada y ejecutarlo en la misma sin problemas.

3.2 Eficiencia

La eficiencia de la librería era un aspecto muy importante. Dado que uno de sus objetivos era permitir el análisis de altos volúmenes de tráfico, era esencial utilizar un diseño que no tuviese un costo elevado de memoria ni de ciclos de *CPU*.

Para lograr una implementación que cumpliera con los parámetros de eficiencia elegidos, se analizaron diferentes diseños para la librería. En todos ellos se tuvieron en cuenta diferentes aspectos que tendrían un impacto en la *performance* de la misma:

- Los campos que no ocupan un número entero de bytes pueden traer problemas durante su implementación, dependiendo del diseño que se utilice. Se puede optar por utilizar formas abstractas de representación de los mismos, que involucren un *overhead* mayor en ejecución pero sean simples de implementar. O bien utilizar mecanismos que tengan un *overhead* nulo en ejecución, pero sea tedioso implementarlos.

- La abstracción que se utilice para representar los diferentes *PDU* también será un factor determinante en la eficiencia de la librería. Ver un paquete como un gran bloque fijo de bytes, adonde cada *PDU* es un *offset* dentro de él es muy eficiente, pero puede ser muy difícil de implementar si se quiere dar soporte a la creación o modificación de paquetes. Por otro lado, la representación de estos como diferentes entidades independientes es mucho mas simple, pero implica un número mucho mayor de alocaciones de memoria, lo que afectará la *performance*.

3.3 Dependencias

La mayoría de los proyectos suele tener dependencias externas, como librerías de terceros que faciliten ciertas tareas. Luego de analizar los requerimientos del proyecto, se identificaron sus necesidades y se decidió la tecnología a utilizar para cumplirlas. Las mismas se encuentran listadas en las siguientes secciones.

3.3.1 Captura de paquetes

La necesidad básica de la librería desarrollada era que pudiese capturar paquetes de red para luego interpretarlos como corresponda. Para poder capturar paquetes, existen diferentes alternativas:

- Utilizar *sockets* crudos: Esta es la solución más difícil y menos portable. Cuando se abre un *socket TCP* o *UDP*, el sistema operativo es el que encapsula los datos que enviamos en uno o más paquetes y realiza mecanismos de control en el caso que aplique. Por otro lado, al utilizar *sockets* çrudos", se pueden enviar y recibir paquetes en sí (incluyendo sus cabeceras de capa de enlace, red y transporte) y no simplemente datos. Este mecanismo es muy tedioso, dado que no es portable, por lo que habría que utilizar directivas para se compilen diferentes fragmentos de código dependiendo del sistema operativo que se use. No sólo eso, sino que además estos no proveen ningún mecanismo simple que permita filtrar tráfico en tiempo real.

- Utilizar *libpcap*: Esta librería es prácticamente un estándar a la hora de capturar paquetes. Además de permitir capturarlos, provee mecanismos para hacer filtrado de paquetes a nivel *kernel*, listar dispositivos de red, poner interfaces en modo monitor y promiscuo. El filtrado de paquetes permite utilizar una sintaxis muy simple y concisa para especificar cuáles se quiere tener en cuenta durante una captura.

Se decidió utilizar *libpcap*, dado que no sólo simplifica la tarea de capturar paquetes enormemente, sino que además está disponible en todos los sistemas operativos a los que se quería dar soporte.

3.3.2 Build system

En este caso y como consecuencia del lenguaje de programación elegido, además, era necesario elegir un mecanismo de compilación o *build system*. Dado que se tenía un conocimiento básico del *GNU build system* y que este es uno de los más utilizados, se lo eligió para cumplir este objetivo.

GNU build system, también conocido como *autotools*, es un conjunto de herramientas que permiten, entre otras cosas, que la compilación sea portable entre diferentes sistemas operativos. Esta apuntado principalmente a sistemas compatibles con *UNIX*, lo que dejaría de lado a *Windows*, pero existen versiones de estas herramientas que han sido portadas a este sistema operativo.

3.4 Portabilidad

Como fue planteado inicialmente, se esperaba que la librería pudiese ser usada en múltiples sistemas operativos. Como consecuencia, era deseable evitar

utilizar funciones que fuesen específicas de algún sistema operativo. Lamentablemente, como a veces se necesita consultar información de dispositivos de red u otros datos, esto puede ser inevitable.

Por consecuencia, cuando esto ocurriese era necesario buscar funciones equivalentes en los demás sistemas operativos, y escribir el código de manera que utilice unas u otras dependiendo de adonde se esté compilando la librería. Esto puede lograrse utilizando el preprocesador de *C++* mediante directivas que permitan habilitar y descartar partes del código dependiendo de la plataforma sobre el que se está compilando.

Por ejemplo, una forma común de permitir que un fragmento de código sea diferente si se está compilando bajo *Windows* se puede lograr verificando si la *macro WIN32* está definida, como puede apreciarse en la figura 5.

Figura 5: Compilación condicional

```
#ifdef WIN32
    // Esta sección sólo se compilará en Windows
    typedef SOCKET TipoSocket;
#else
    // Esta sección se compilará en cualquier otro S.O.
    typedef int TipoSocket;
#endif
```

Por otro lado, cualquier dependencia que tuviese el proyecto, como por ejemplo otras librerías o el mecanismo de compilación utilizado, debían ser soportadas en todos los sistemas operativos a los que se quería dar soporte. Dado que únicamente se tomó *libpcap* como dependencia, y el mecanismo de compilación utilizado fue *autotools*, ambos siendo multiplataforma, esto no fue un problema.

3.5 Documentación

Otro aspecto importante del diseño fue el de incluir documentación en el proyecto. Muchas veces uno se encuentra con librerías que parecen ser excelentes, pero le cuesta mucho comenzar a usarlas debido a la escasez de documentación disponible. Fue por esto que se hizo un gran hincapié en documentar cada clase y función disponible, indicando sus objetivos y cualquier detalle relevante sobre ellas.

Se decidió utilizar la herramienta *Doxygen*, que permite documentar el código y luego generar documentación en diferentes formatos (*html*, *LaTeX*,

etc).

El mecanismo de definición de la documentación de esta herramienta puede verse en la figura 6. Al correr la herramienta *doxygen* sobre el código, esta analizará las anotaciones y las utilizará al generar la documentación en el formato elegido.

Figura 6: Ejemplo de documentación con Doxygen

```
/**
 * \class Foo
 * \brief Descripción breve de la clase
 *
 * Descripción más extensa de la clase.
 */
class Foo {
public:
    /**
     * \brief Esta es una descripción breve del método.
     *
     * Esta es una descripción más larga del método.
     * Acá se podrán explicar detalles internos del
     * mismo, entre otras cosas.
     *
     * \param x Indicamos para qué sirve el
     * parámetro x.
     * \return Indicamos que retorna la función.
     */
    int bar(int x);
};
```

Además, se crearía un sitio que incluyese información acerca de la librería, como ejemplos y tutoriales además de la documentación generada. De esta forma, cualquiera podría tener acceso a recursos que le permitan aprender a utilizarla sin problemas.

3.6 Licencia

La librería debía ser de código abierto, para que cualquiera pudiese utilizarla en sus proyectos. Como consecuencia, se debió elegir una licencia para el proyecto que permitiese su libre uso y distribución.

Inicialmente se utilizó la licencia *LGPL*[24] o *GNU Lesser Public Licence*. El problema con ésta es que su uso comercial es un poco restrictivo, forzando a cualquiera que la utilice a licenciar su código en *LGPL* o bien hacer pública cualquier modificación que le realice. Como lo que se buscaba era que cualquier pudiese utilizarla y no se restringiera su uso en absoluto, se la terminó cambiando.

Finalmente se utilizó la licencia *BSD-2*[25]. Esta obliga únicamente a:

- Que cualquier distribución del código fuente mantenga la nota de *copyright* que se incluya originalmente en el proyecto.
- Que cualquier distribución de la librería en formato binario, es decir compilada, también mantenga esa nota.

De esta manera, cualquiera puede utilizarla y modificarla para uso tanto de código abierto como comercial, siempre y cuando se distribuya el *copyright* original.

4 Capítulo 4: Diseño de la librería

Como se mencionó en secciones anteriores, el diseño de la librería fue principalmente guiado por la necesidad de que ésta fuese rápida. Fue por esto que se analizaron diferentes diseños y formas de estructurar la librería. Se hizo hincapié mayormente sobre los siguientes aspectos:

- Estructuras para representar los campos de cada *PDU*.
- El modo de enlazar los *PDU* que conforman un paquete.
- El mecanismo para capturar paquetes.

En las siguientes secciones se plantearán los diferentes diseños analizados y elegidos para cada característica de la librería.

4.1 Representación de campos: utilizando abstracciones

El primer diseño en el que se pensó para representar los campos de cada *PDU* fue uno que simplificara la implementación de nuevos protocolos. Esto podría lograrse utilizando abstracciones para cada campo de los *PDU*, tal como se muestra en la figura 7.

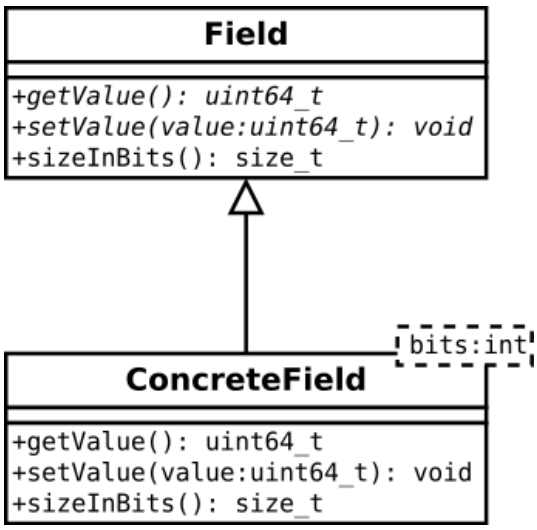

Figura 7: *Diseño abstracto*

La clase *ConcreteField* debería ser instanciada con una cantidad de *bits* fija. Esta implementación permitiría realizar 3 operaciones:

- Asignarle un valor al campo.
- Consultar el valor del campo.
- Consultar la longitud del campo en *bits*.

De esta manera, la estructura de cada protocolo estaría representada por un vector de punteros a *Field*s, donde cada uno de esos es en realidad una instancia de *ConcreteField*. La figura 8 muestra un ejemplo de como se podría representar una estructura utilizando este esquema.

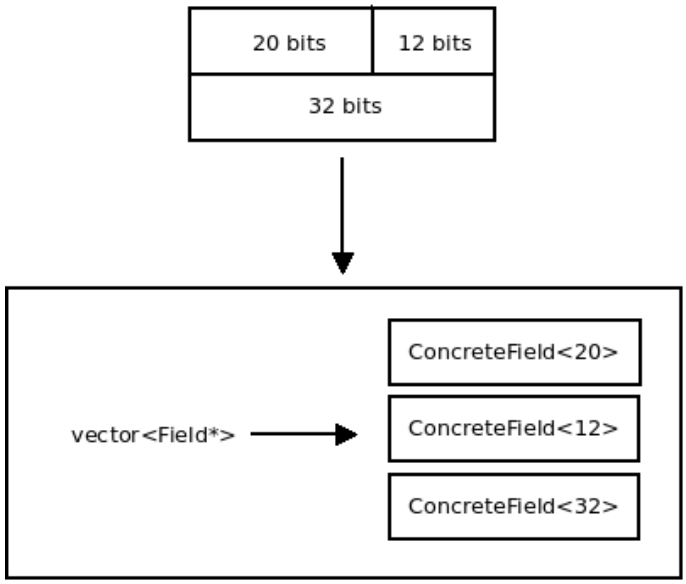

Figura 8: *Ejemplo de un PDU representado a través de las clases Field y ConcreteField*

4.1.1 Ventajas

- Es muy simple implementar nuevos protocolos. Esto se debe a que al ser completamente abstracto, se podrían implementar *scripts* que dada una lista de tuplas *(nombre de campo, longitud)* que identifique las propiedades de cada campo, nos genere la implementación completa de la clase que represente el *PDU*.
- Lograr que la implementación funcione sobre diferentes arquitecturas también es muy simple. Como cada valor esta guardado en una enti-

dad diferente, lograr que estos sean serializados correctamente y en el *endian* apropiado no trae complicaciones.

4.1.2 Desventajas

- *Overhead* de tamaño: Para representar cualquier campo, sea cual sea su longitud, se tiene un *overhead* muy grande. A continuación se dará un ejemplo, suponiendo que estamos sobre una arquitectura en la que los punteros ocupan *8 bytes*, como puede ser una arquitectura de *64 bits*.

  Dado que no es posible definir tipos de datos cuya longitud no sea un número entero de bytes, siempre se necesitarán varios *bits* extras para definir cada campo. Un ejemplo extremo sería definir un campo de 1 *bit*, para el cual necesitaremos un campo de *8*.

  Las clases que presentan polimorfismo en ejecución a través de herencia en *C++* suelen ser implementadas internamente por el compilador usando un puntero a una tabla de punteros a funciones, denominada *vtable*. Ésta permite que al llamar a una función de un objeto polimórfico, se elija la apropiada dependiendo del tipo del mismo. Como consecuencia, cada objeto polimórfico tiene implícitamente un puntero a esta estructura. Como cada estructura que representa a un campo tiene estas características, cada uno de éstos ocuparán *8 bytes* más.

  Como resultado, el *overhead* de tamaño en este diseño es muy grande. Tomando como ejemplo la representación de un campo de *1 bit*, sería necesario:

  - Un campo de *1 byte* para representarlo.
  - Un puntero a la *vtable* mencionado en el punto anterior, el cual sería de *8 bytes*.
  - Un puntero al campo en sí, que estará almacenado dentro del vector de campos, es decir otros 8 *bytes* más.

  Por lo tanto necesitaríamos *136 bits* para representar un campo de uno. Esto es demasiado, teniendo en cuenta que los *PDU* suelen estar compuestos de varios campos de tamaños chicos, agregando un *overhead* indeseable.

- Alocaciones dinámicas: Al utilizar esta representación, cada *ConcreteField* debe ser alocado en memoria dinámica ó *heap*. Este tipo tiene un costo mucho mayor a una simple alocación en la pila de ejecución.

Además, dado que los campos serán de tamaño chico, esto generará una gran fragmentación de la memoria.

- El análisis y serialización de los *PDU* tiene su costo: Al querer analizar un bloque de datos y construir el *PDU* correspondiente a partir de él, habría que hacer un procesamiento adicional. Esto se debe a que tendría que irse consumiendo el bloque de datos de a poco, extrayendo el valor de cada campo siempre teniendo en cuenta su orden y longitud.

  De la misma manera, al serializar un *PDU* deberíamos realizar la operación inversa: recorrer la secuencia de campos, extrayendo sus valores individuales y escribiéndolos como una secuencia de *bytes*.

Como puede verse, esta solución tiene varios aspectos negativos. Si bien sería muy buena para facilitar la implementación de nuevos protocolos, traería un impacto indeseado en la *performance* de la librería. Fue por esto que se terminó descartando y se pensó otro diseño que no tuviese estas desventajas.

4.2 Representación de campos: utilización de *bitfields*

Otro diseño analizado para la representación de las estructuras de cada *PDU*, que terminó siendo el elegido, involucraba la utilización de *bitfields*. Los *bitfields* se utilizan para definir estructuras compuestas por varios campos contiguos cuyas longitudes individuales se definen en cantidad de *bits* y no *bytes* como se realiza normalmente.

Un ejemplo de la definición de una estructura mediante la utilización de *bitfields* puede verse en la figura 9, así como su representación gráfica se encuentra en la figura 10.

Figura 9: Estructura definida con *bitfields*

```
struct bitfield_example {
    // Un campo de 24 bits seguido de uno de 8.
    uint32_t f1:24,
             f2:8;
    // Campos de 13 y 3 bits
    uint16_t f3:13,
             f4:3;
    // Campos de 5 y 3 bits
    uint8_t f5:5,
            f6:3;
```

```
    // Campo de 8 bits9
    uint8_t f7;
};
```

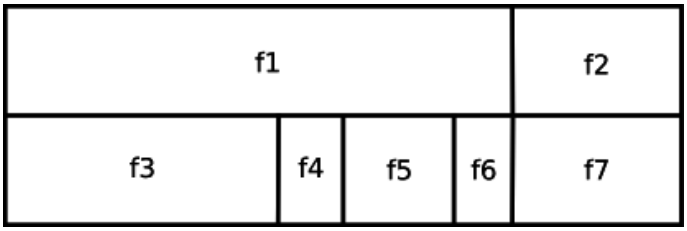

Figura 10: *Representación gráfica de la estructura definida con bitfields*

Internamente, el compilador representa cada lectura o escritura sobre esos campos con operaciones con *bits* utilizando máscaras y *bit shifts*, permitiendo que estas operaciones sean transparentes para el programador.

Para que el compilador respete los tamaños de los campos y no agregue un *padding* al final de la estructura, se tiene que utilizar una directiva especial, la cual no es estándar, por lo que va a depender de cada uno.

En el compilador de *GNU*, *gcc*, o compiladores compatibles, como *clang*, esto se puede lograr agregando un atributo a la estructura, tal como lo muestra la figura 11.

Figura 11: Packing en compiladores gcc-compatibles

```
struct bitfield_example {
    // Definición
} __attribute__((packed));
```

Por otro lado, en el compilador de *Microsoft*, *VC++*, esto se logra utilizando una directiva que indica que las estructuras que sigan utilizarán un *packing* a *1 byte* y luego otra para quitar este efecto a todas las definiciones que sigan. Un ejemplo de esto puede verse en la figura 12.

Figura 12: Packing en Microsoft VC++

```
__pragma( pack(push, 1) )
struct bitfield_example {
    // Definición
} __pragma( pack(pop) );
```

De esta manera, las estructuras que representen a los *PDU* se pueden definir con campos que tengan el tamaño exacto que se necesite, sin desperdiciar ni un *bit*.

4.2.1 Incompatibilidad de *endianness*

Una de las ventajas de este diseño es poder analizar y serializar *PDU*s con prácticamente costo cero. Esto se puede lograr simplemente leyendo o escribiendo el contenido de la estructura que definimos sin hacer ningún análisis o computación de los campos individuales que lo componen.

Para lograr esto, hay que tener en cuenta la incompatibilidad que se introduce al definir estructuras que luego sean serializadas e interpretadas utilizando otro *endian*. Esto se debe a que al definir una estructura y luego serializarla como un bloque de *bytes*, la salida será diferente dependiendo del *endian* de la maquina sobre el que se corra el código.

La figura 13 muestra un ejemplo de una estructura que contiene este problema. Al serializarla en una arquitectura *little endian*, el *bit b7* estará al comienzo y el *b0* al final. Por otro lado, en una arquitectura *big endian* será lo opuesto.

Figura 13: Ejemplo de estructura cuya representación depende del *endian*

```
struct foo {
    uint8_t b0:1,
            b1:1,
            b2:1,
            b3:1,
            b4:1,
            b5:1,
            b6:1,
            b7:1;
};
```

Como consecuencia de esto, para poder definir estructuras utilizando *bit-fields*, es necesario usar compilación condicional para utilizar una secuencia de campos diferente dependiendo de la arquitectura sobre la que se compile.

4.2.2 Ventajas

- Cero *overhead* de análisis y serialización: al tener una estructura compuesta por campos contiguos y cada uno es de exactamente el tamaño

necesario, analizar un bloque de *bytes* o serializar esta estructura se reduce a una escritura de memoria. Además de que esto involucra pocos ciclos del procesador, también es muy simple de implementar.

- Cero *overhead* de tamaño al representar *PDU*s: como los campos tienen asignada una longitud en *bits*, la estructura que se defina va a ocupar exactamente la cantidad de *bytes* que debe ocupar.

4.2.3 Desventajas

- Dificultad en la definición de estructuras: para cada *PDU* que se quiera implementar, es necesario analizar bien cada campo que lo compone y definir una estructura apropiada tanto para *big* como para *little endian*. Esto es bastante tedioso, dado que encontrar el orden correcto de los campos no es tan intuitivo como parece.

Finalmente se optó por utilizar *bitfields* para representar a las estructuras de los diferentes protocolos. Era preferible hacer que el desarrollo de la librería fuese más tedioso, pero que como consecuencia esta fuese lo más rápida posible.

4.3 Representación de *PDU*s

Cada paquete puede imaginarse como una sucesión de capas apiladas, adonde cada una pertenece a algún protocolo en particular. Éstas fueron modeladas con objetos de tipo *PDU* que tienen atributos y operaciones válidas para cada protocolo.

Además, como cada capa puede contener otra que la suceda, se pensó a un paquete como una lista enlazada de *PDU*s, adonde cada una de ellas puede tener a lo sumo un sucesor. Como consecuencia, no se creó una entidad "paquete", sino que un paquete sería una cadena de uno o más *PDU*s.

A pesar de que todos los protocolos tienen estructuras diferentes, si se imagina a cada *PDU* como una entidad, todas tienen algunas operaciones en común. Entre ellas, se podría pensar en las siguientes:

- Construirla a partir de un bloque de *bytes*.
- Consultar el tamaño en *bytes* que ocuparía si fuese serializada.
- Serializarla a un bloque de *bytes*.
- Obtener el tipo de *PDU* que representa.

- Obtener el *PDU* que le sigue.
- Dado un tipo de *PDU*, buscar entre las capas que le siguen, la primera que tiene el mismo tipo. Por ejemplo, dado un paquete cualquiera, buscar el objeto correspondiente a un segmento *IP*.

Utilizando esta lista se creó la clase *PDU*, que serviría de clase base para todos los protocolos que se implementen. La figura 14 muestra una interfaz reducida de la clase, obviando varios detalles específicos de *C++* (por ejemplo las palabras clave *virtual* y *const*) que sí se tuvieron en cuenta en la implementación real.

Figura 14: Interfaz de la clase *PDU*

```
class PDU {
public:
    // Los tipos de PDU que existen
    enum PDUType {
        ETHERNET,
        IP,
        TCP,
        // ...
    };

    // Tamaño en bytes
    size_t size();

    // Serializarlo a un vector de bytes
    vector<uint8_t> serialize();

    // Tipo de PDU
    PDUType pdu_type();

    // Setter para el siguiente PDU
    void inner_pdu(PDU* next);

    // Getter para el siguiente PDU
    PDU* inner_pdu();

    // Permite buscar PDUs hijos
    template<typename Type>
```

```
    PDU& rfind_pdu();

    template<typename Type>
    PDU* find_pdu();
};
```

Luego, cualquier protocolo nuevo que se quiera agregar debe heredar de esta clase e implementar los métodos que correspondan.

Quizás los métodos más útiles de estos, desde el punto de vista de un usuario de la librería, son *rfind_pdu* y *find_pdu*. Con estos se pueden encontrar las diferentes capas de un paquete y así poder procesar sus datos. En la figura 15 puede verse un ejemplo de su utilidad.

Figura 15: Uso del método *rfind_pdu*

```
void process(const PDU &pdu) {
    // Buscar la capa IP
    const IP& ip = pdu.rfind_pdu<IP>();

    // Buscar la capa TCP
    const TCP& tcp = pdu.rfind_pdu<TCP>();

    // Imprimimos IP y puerto origen y destino
    std::cout << ip.src_addr() << ':' << tcp.sport()
              << " -> " << ip.dst_addr() << ':'
              << tcp.dport() << std::endl;
}
```

4.4 Serialización de paquetes

Para poder serializar un paquete correctamente, hay que tener en cuenta algunas características que tienen en común varios protocolos:

- *Checksums*: Los *PDU* que usan chequeos de errores utilizan un algoritmo sobre el contenido de las capas superiores a él, el cual genera un número, normalmente de 16 *bits*, y lo guardan dentro de un campo de su estructura. Como consecuencia, las capas deben serializarse en orden inverso: primero la capa de aplicación, y por último, la de enlace.
- *Checksums* dependientes de capas inferiores: algunos protocolos, como *TCP*, generan un *pseudo-header* utilizando datos de la capa de red,

como dirección *IP* origen y destino, que incorporan al cálculo de su *checksum*. Esto implica que, si bien los protocolos de capas superiores se serializan antes que los de las inferiores, en caso que algún campo se inicialice automáticamente durante la serialización, esto tiene que hacerse antes que cualquier otra cosa.

- Utilización del identificador de la siguiente capa: cada protocolo, salvo los de capa de transporte y aplicación, utilizan algún tipo de identificador que permite saber de qué tipo es la capa que sigue. De esta manera, al analizar paquetes se puede saber como interpretarlos. Esto se tiene que detectar al momento de la serialización y se tiene que buscar el identificador apropiado y guardarlo en el campo que corresponda.

- *Trailers*: algunos protocolos como *Ethernet* o *RadioTap* insertan datos al final del paquete. Esto puede ser un *checksum* o simplemente *padding*. De cualquier forma, hay que diferenciar ésto de lo que es tamaño de la cabecera de un *PDU*. Este último influirá en el *offset* sobre el que se escriban los protocolos de las capas que sigan, mientras que el tamaño del *trailer* tiene que ser tomado en cuenta para la longitud total del paquete, pero no tendrá influencia en ese desplazamiento.

Un ejemplo del segundo punto mencionado arriba, que implicó modificaciones en el código es la dirección origen en el protocolo *IP*. *libtins* permite generar y enviar paquetes cuya capa inferior sea de red, ya que en ese caso la de enlace será creada automáticamente por el *kernel*. Además, este asignará de la misma manera la dirección *IP* origen apropiada, dependiendo de cual sea la interfaz por la que vaya a ser enviado el paquete. Pero esto no nos sirve, porque si bien la *IP* origen del paquete luego de ser enviado va a ser correcta, el *checksum* de *TCP* depende de ella y se calcula antes del envío. Es por esto que, antes de serializar, si se detecta que el protocolo de capa de red es el primero, se infiere esta dirección y se asigna a ese campo.

Teniendo éstos items en cuenta, el algoritmo de serialización constó de los siguientes pasos:

1. Extraer el tamaño total del paquete y alocar un bloque de *bytes* suficientemente grande como para contenerlo.

2. Propagar una acción de "preparación pre-serialización". En ésta, los protocolos de capa de red buscan y asignan su dirección origen si son la capa inferior del paquete.

3. Propagar la serialización, de manera que primero se efectúe sobre los protocolos de capas superiores. Al procesar la capa *X*, ésta debe escribirse sobre el *offset* en el que se encontraría la *X - 1* sumándole el tamaño de la cabecera de este último.

La figura 16 muestra como se implementó este método, obviando algunos detalles menores de implementación interna.

Figura 16: Ejemplo de serialización

```
void PDU::serialize(uint8_t *buffer, uint32_t total_sz)
{
    // header_size devuelve el tamañi de la cabecera.
    // trailer_size es del trailer
    uint32_t tamaño_pdu = header_size()+trailer_size();

    // Propaga la preparación pre-serialización.
    prepare_for_serialize();

    // Si hay un PDU de la capa siguiente
    if(pdu_que_sigue != NULL) {
        // Serializarlo
        pdu_que_sigue->serialize(
            buffer + header_size(),
            total_sz - tamaño_total_pdu
        );
    }

    // Serializa este PDU. En este punto,
    // todas las capas superiores ya fueron serilizadas
    write_serialization(buffer, total_sz);
}
```

4.5 Interpretación de paquetes

El mecanismo de interpretación de paquetes es bastante simple. La posible complejidad depende de cada protocolo y su estructura interna. En general, para interpretar un paquete se realizan los siguientes pasos:

1. Se comienza por el primer tipo de *PDU* que contenga el paquete. Si

se están capturando paquetes, este se puede encontrar usando la *API* de *libpcap*.

2. Se construye un objeto del tipo que se corresponde con el *PDU* del protocolo de capa de enlace que contenga el paquete capturado. Éste internamente va a hacer una copia de la estructura principal del protocolo y a copiar cualquier dato opcional que pueda tener.

3. Si a continuación quedan datos sin procesar, se analiza el identificador del tipo del siguiente *PDU*. En caso de que éste sea de un protocolo implementado, se procede a construir un objeto de ese tipo utilizando el resto de los datos y se vuelve al paso 3. En caso contrario, se construye un objeto *RawPDU* que simplemente representa un bloque de *bytes* y se termina la interpretación. En cualquiera de los dos casos, el objeto construido pasará a ser el *PDU* hijo del que se está construyendo.

Como resultado, se tendrá una cadena de objetos *PDU*, cada uno del tipo que se corresponda con el protocolo del paquete original.

4.6 Representación de opciones *TLV*

Como se mencionó en la sección anterior, una característica en común entre varios protocolos (como *IP*, *TCP* y *DHCP*, entre otros) es que tienen varios campos opcionales, todos codificados usando la representación *TLV* o *type-length-value*. Para no reimplementar esta estructura una vez por cada protocolo que la usara, se creó una entidad a parte.

La estructura básica de la clase que inicialmente se creó con este objetivo puede verse en la figura 17.

Figura 17: Definición de la clase *PDUOption*.

```
// Algunos PDU usan identificadores de 8 bits,
// otros de 16. Por eso usamos una clase template,
// que permite elegir el tipo que se va a usar
// al instanciar la clase.
template<typename OptionType>
class PDUOption {
public:
    // Getters y setters
private:
    // El tipo de la opción
    OptionType tipo_de_opcion;

    // Guardamos el campo de tamaño aparte,
    // para poder utilizar longitudes diferentes
    // a las del valor verdadero. Esto es útil para
    // hacer fuzzing
    uint16_t campo_tamaño;

    // El valor de la opción.
    std::vector<uint8_t> valor;
};
```

Esta implementación tiene algunos problemas, principalmente:

- Como la clase *vector* aloca sus datos en memoria dinámica, cada vez que se instancie una opción se estará realizando una alocación de memoria.
- Si bien no es un comportamiento requerido por el estándar de *C++*, la clase *vector* suele pre-alocar almacenamiento para una algunos elementos. Esto no sería un problema, salvo porque existen opciones en casi todos los protocolos que son utilizadas principalmente como *padding*. Esto implica que tendremos muchas opciones que van a estar presentes, pero que no acarrearán datos. Como consecuencia, cada opción implicará una alocación de memoria para ese vector, incluso si no acarrea datos más que su tipo y longitud.

Una mejora a esta clase surgió de analizar los datos que normalmente se ven en las opciones. Es muy común en protocolos como *TCP* que éstas

transporten datos muy pequeños, normalmente de unos pocos *bytes*. Esta característica puede aprovecharse utilizando la misma técnica que suelen utilizar algunas implementaciones de la clase *string* de la librería estándar de *C++*: *small-string-optimizacion* o *SSO*.

La idea es que, como muchas veces los datos que se van a guardar son chicos, se puede transportar un pequeño lugar de almacenamiento que no utilice la memoria *heap*. Si la cantidad de datos que se guardan es pequeña, se almacenan ahí. En caso contrario, se hace una alocación dinámica.

Además, la idea de la técnica es comprimir al máximo la clase. Es por esto que se hace uso de una *union* para guardar tanto un puntero a la memoria dinámica que se pueda estar utilizando, como el *buffer* para datos pequeños. Las *union* permiten guardar múltiples valores en la misma dirección de memoria (básicamente interpretándola de formas diferentes), por lo que son muy útiles en este caso. La figura 18 muestra como podría implementarse esta clase.

Figura 18: Definición de la clase *PDUOption* mejorada.

```
template<typename OptionType>
class PDUOption {
public:
    // El tamaño del buffer chico
    static const size_t tamaño_buffer_chico = 8;

    // Getters y setters
private:
    OptionType tipo_de_opcion;

    // Tamaño real del valor de la opción.
    uint16_t tamaño_real;

    // Tamaño del campo "longitud".
    uint16_t tamaño_del_campo;

    // El valor es una union.
    union {
        // Buffer pequeño para opciones chicas.
        uint8_t buffer_chico[tamaño_buffer_chico];

        // Puntero a memoria dinámica.
```

```
        uint8_t* puntero_a_memoria_dinamica;
    } valor;
};
```

Como resultado, se tiene una clase que guardará los datos sin realizar alocaciones mientras estos sean chicos, y recién cuando sean más grandes se realizará una alocación de memoria dinámica. En este caso se utilizó un tamaño de 8 *bytes* para el tamaño de este almacenamiento porque es suficiente para guardar la mayoría de los datos de las opciones de distintos protocolos y además es el mismo tamaño que suelen tener los punteros en arquitecturas de 64 *bits*, por lo que si se usa una con esta características no se desperdiciará nada de tamaño.

Usando pruebas que analizaran la velocidad de análisis de muchos paquetes *TCP* con 5 opciones con tamaños menores a 8 *bytes*, esto trajo una mejora en velocidad de aproximadamente 30 %.

4.7 Captura de paquetes

Se necesitaba agregar algún mecanismo para capturar paquetes de red y leerlos de archivos *pcap* utilizando las funciones que provee la librería *libpcap*. Como ésta maneja prácticamente igual la captura de paquetes desde un dispositivo y desde un archivo, se creó una jerarquía de clases. Esta estaría compuesta por una clase base, *BaseSniffer* que contuviese los mecanismos en común entre los tipos de captura, y dos subclases *Sniffer* y *FileSniffer*, cada una implementando los que diferían.

Se crearon diferentes mecanismos para procesar los paquetes que se capturen, cada uno con diferentes características.

4.7.1 Captura de a un paquete

Uno de los mecanismos de captura que se creó fue el que permite pedirle los paquetes uno a uno a un objeto *BaseSniffer*. Cada vez que se llama al método *next_packet*, esto puede retornar un puntero a un *PDU* que contenga el último capturado, o un puntero nulo en caso de que la fuente se haya agotado o se haya encontrado un error.

La desventaja de esta técnica es que al retornar punteros, se tiene que hacer manejo explícito de memoria o recurrir al uso de punteros inteligentes. La figura 19 muestra un ejemplo de como se pueden procesar paquetes usando este método.

Figura 19: Ejemplo de captura usando *BaseSniffer::next_packet*

```
// Creamos el objeto Sniffer
Sniffer sniffer("eth0");

// Leemos paquetes hasta que la fuente se agote.
// Usamos un puntero inteligente que desalocará
// el PDU cada vez que se termine de utilizarlo.
while(unique_ptr<PDU> pdu{ sniffer.next_packet() }) {
    // usar pdu
}
```

4.7.2 Captura usando un *callback*

Otra de los mecanismos de captura fue el de permitir que se provea una función función *callback*, la cual es llamada una vez por cada paquete que se capture. En esta, el manejo de memoria es automático por parte del objeto *BaseSniffer*, por lo que no hay que preocuparse por fugas de memoria.

La función *callback* puede no sólo ser una función libre, sino de cualquier tipo que soporte el operador *()*. Esto permite llamar métodos de instancia de alguna clase utilizando objetos *proxy*, por lo que es muy útil.

La única restricción de esa función es que debe:

- Tomar algún tipo de referencia a un *PDU* como parámetro.
- Retornar un valor booleano que indique si se quiere seguir capturando paquetes o se debe cortar el bucle de captura.

La figura 20 muestra un ejemplo de la utilización de este método.

Figura 20: Ejemplo de captura usando *BaseSniffer::sniff_loop*

```
// La función callback
bool callback(PDU& pdu)
{
    // Procesar el PDU
    // .....

    // Retornamos true para que siga capturando
    return true;
}
```

```
// Creamos el objeto Sniffer
Sniffer sniffer("eth0");

// Iniciamos la captura llamando a callback
sniffer.sniff_loop(callback);
```

4.8 Análisis automático de las capas de un paquete

Al analizar tráfico, era de esperarse que los paquetes que se capturaran fuesen analizados automáticamente por la librería, y separados en diferentes objetos que representen a cada capa. Algo que tuvo que plantearse al diseñarla fue hasta qué capa del modelo *TCP/IP* se realizaría esta interpretación.

Una opción era permitir que todos los protocolos conocidos fuesen interpretados automáticamente. Por ejemplo, si se capturaba una consulta *DNS* sobre *UDP*, esto nos generaría una estructura de objetos similar a la que muestra la figura 21.

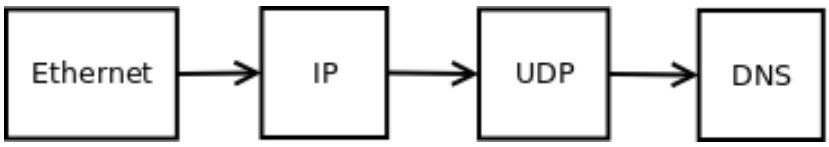

Figura 21: *Ejemplo de un paquete DNS con todas sus capas interpretadas*

Esto es ventajoso porque no habría que hacer nada adicional para trabajar con las propiedades de la capa de aplicación que posiblemente pueda tener cada paquete. Pero por otro lado, también trae varias desventajas:

- Costo adicional: analizar los protocolos de capa de aplicación obviamente implica más procesamiento que si no se interpretara automáticamente. Además, podría ocurrir que algún usuario de la librería no quiera trabajar con esta capa, por lo que estaríamos consumiendo ciclos de *CPU* sin sentido.

- Dificultad de detección de protocolos: puede ser difícil detectar qué protocolo de capa de aplicación se está utilizando en un paquete. La capa de transporte no tiene ningún campo que nos indique cuál es el protocolo usado en la de aplicación, como pasa en las anteriores, por lo que habría que .ªdivinarçual es el utilizado. En el caso de *DNS*, uno podría

asumir que si un paquete *TCP* o *UDP* tiene puerto destino u origen 53, entonces seguramente es una consulta o respuesta *DNS*, pero no puede estar seguro de que realmente lo sea.

Es peor en el caso de otros protocolos, como *HTTP*, que pueden aparecer en cualquier puerto, lo que hace que su detección sea más compleja y además su contenido puede estar fragmentado en múltiples paquetes.

Por estas desventajas, se terminó optando por sólo analizar automáticamente hasta la capa de transporte. Los datos que se envíen sobre ésta serían encapsulados en un objeto que represente un bloque de datos, dándole al usuario la librería interpretar ese bloque como cualquier otro protocolo implementado. Volviendo al ejemplo de *DNS*, un paquete se vería como la estructura mostrada en la figura 22.

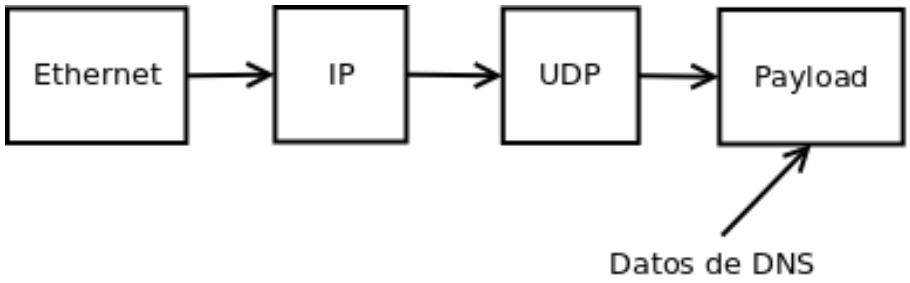

Figura 22: *Ejemplo de un paquete DNS sin interpretar los datos de capa de aplicación*

4.9 *Testing*

Fue indispensable definir *tests* de unidad que comprobaran que todo lo que se implementó funcionara correctamente. Esto fue además muy útil a la hora de portar *libtins* a otros sistemas operativos y arquitecturas.

Se eligió el *framework googletest* para crear las diferentes pruebas. Éste trae diferentes clases y *macros* que permiten definir diferentes *tests* de una manera simple y rápida. La figura 23 muestra un ejemplo de la definición de un bloque de pruebas utilizando *googletest*.

Figura 23: Ejemplo de *tests* creados con *googletest*

```
// Inclusión de los headers de googletest
#include <gtest/gtest.h>

// Clase que define un bloque de tests
class FooTest : public testing::Test {
public:

};

// Definicion de un test
TEST_F(FooTest, Test1) {
    // Esto hará que el test falle si 1 + 1 no es 2
    EXPECT_EQ(2, 1 + 1);

    // Fallará si la expresión 5 < 7 no es true
    EXPECT_TRUE(5 < 7);
}

// Otro test
TEST_F(FooTest, Test2) {
    // ...
}
```

Se implementaron pruebas para cada clase que representaba un *PDU*, probando principalmente los siguientes aspectos:

- *Setters* y *getters*: luego de asignarle un valor a algún atributo de un objeto, si se consultara qué valor tiene, esto debería retornar lo mismo que se le asignó. Esto puede parecer una prueba sin mucho sentido, pero dado que en los *getters* y *setters* de cada clase se cambiaba el *endian* (tal como se describe en la sección 4.11), esto fue indispensable y llevó a que se encuentren una gran cantidad de errores en el código.
- Construcción de un *PDU* a partir de un bloque de *bytes*: se capturaron paquetes de red y extrajeron los datos correspondientes al protocolo que se quería probar. Luego se construyó un objeto del tipo que representase a ese *PDU* a partir de esos *bytes* y se verificó que sus atributos tuviesen los mismos valores que los del paquete original.

- Serialización de un *PDU*/paquete: se verificó que al construir un paquete a partir de un bloque de *bytes* y luego serializarlo, el resultado fuese el mismo bloque de *bytes* original.

4.10 Portabilidad entre sistemas operativos

Como se mencionó en secciones anteriores, la portabilidad de la librería era un aspecto muy importante que se tuvo que tener en cuenta al diseñarla. No sólo se quería que ésta fuese utilizable en diferentes sistemas operativos, pero también en diferentes arquitecturas.

Lograr que la librería funcione en diferentes sistemas operativos resultó ser más difícil de lo que se esperaba. Incluso para que funcione entre sistemas *POSIX*-compatibles, hubo que hacerle varias modificaciones debido a que algunos mecanismos funcionaban de formas diferentes.

Principalmente, el mecanismo de envío de paquetes es muy diferente entre todos los sistemas a los que se le quiso dar soporte. Se tiene que tener en cuenta que en este caso se podría haber utilizado la librería *libpcap* para realizar el envío, lo que hubiese facilitado enormemente este mecanismo. Sin embargo, esta sólo permite enviar paquetes que incluyan protocolos de capa de enlace, no permitiendo enviar aquellos que contienen uno de capa de red como la inferior. Como consecuencia, se implementó el mecanismo manualmente utilizando *sockets* crudos, para darle una mayor flexibilidad al usuario.

Por otro lado, algunas utilidades que se le agregaron, como la de permitir consultar la tabla de ruteo de la máquina y así poder decidir por qué interfaz enviar paquetes, o incluso la de listar interfaces de red, también debieron ser implementadas de formas diferentes.

Luego de terminar de portar la librería, se probaron todos los *tests* implementados en cada sistema operativo y arquitectura para garantizar que todos pasen correctamente. Esto se realiza regularmente cada vez que se introduce un cambio que podría traer problemas en algún sistema (como la implementación de un nuevo protocolo).

4.10.1 GNU/Linux

Inicialmente se comenzó desarrollando la librería en *GNU/Linux* y apuntando a que primero funcione en este sistema operativo para luego portarlo al resto. Es por esto que a continuación se indicarán qué mecanismos se utilizaron para realizar las mismas tareas que luego tuvieron que ser modificados utilizando compilación condicional para que funcionen en otros

sistemas operativos.

- Mecanismo de envío de paquetes de capa de enlace: este mecanismo es bastante simple en *GNU/Linux*. Básicamente se tiene que abrir un *socket* de tipo *PF_PACKET* y luego utilizar la *system call sendto* para enviar cada paquete.

  Además para cada paquete que se quiera enviar, es necesario construir una estructura de tipo *sockaddr_ll*, la que permitirá indicar que interfaz de red se utilizará y a qué dirección *MAC* se lo enviará.

  Un ejemplo de este mecanismo se muestra en la figura 24.

Figura 24: Apertura de un socket de capa 2 y envío de un paquete en GNU/Linux

```
// Apertura del socket
int l2sock = socket(
    PF_PACKET,
    SOCK_RAW,
    htons(ETH_P_ALL)
);

// Estructura para indicar los datos de envío
struct sockaddr_ll addr;
// La inicializamos en 0
memset(&addr, 0, sizeof(struct sockaddr_ll));
// Familia del socket
addr.sll_family = htons(PF_PACKET);
// Protocolo
addr.sll_protocol = htons(ETH_P_ALL);
// Longitud de la dirección MAC
addr.sll_halen = 6;
// Indice de la interfaz
addr.sll_ifindex = INDICE_DE_INTERFAZ;
// Copiamos la dirección MAC
memcpy(&(addr.sll_addr), DIRECCION_MAC, 6);

// Envío del paquete
sendto(
    l2sock,
    BUFFER_PAQUETE,
```

```
        LONGITUD_PAQUETE,
        0,
        addr,
        sizeof(addr)
    );
```

- Una utilidad que se agregó fue la de permitir consultar la tabla de rutas del sistema operativo. Esto es muy útil para saber, dada una dirección *IP* a la que se quiere enviar un paquete, por qué interfaz debería enviarse.

  La figura 25 muestra la estructura utilizada en *libtins* para representar cada entrada en esta tabla. De esta manera, se implementaría un método que no tomara ningún parámetro y devolviera su estado actual como un vector de *RouteEntry*.

Figura 25: Estructura que define cada entrada en la tabla de rutas

```
struct RouteEntry {
    // Nombre de la interfaz
    std::string interface;

    // IP destino
    IPv4Address destination;

    // Gateway
    IPv4Address gateway;

    // Máscara
    IPv4Address mask;
};
```

En *GNU/Linux* se puede consultar la tabla de rutas leyendo del archivo virtual */proc/net/route*. Este es generado por el kernel cada vez que lo leemos, y devuelve texto plano de fácil análisis, adonde cada linea indica una entrada en la tabla, por lo que no hubo complicaciones al implementarlo.

4.10.2 BSD

Al querer portarla a sistemas compatibles con *BSD*, como *FreeBSD* ó *Mac OSX*, se encontraron algunos problemas que implicaron modificar algunos mecanismos de la librería. Los principales fueron:

- Mecanismo de envío de paquetes: el envío y recepción de paquetes de capa de enlace en *BSD* es muy diferente al utilizado en *GNU/Linux*. Por suerte, la recepción de paquetes ya era implementada por *libpcap*, por lo que no hubo que modificar nada en ese aspecto.

  Para el envío sí hubo que modificar ciertas cosas. Los *sockets raw* en *BSD* no se crean llamando a la *system call socket* como en *GNU/Linux*. En cambio, se tiene que abrir un dispositivo en el directorio */dev/* cuyo nombre comience con "*bpf*"seguido por un número, inicialmente 0. Cada vez que una aplicación abra uno de esos dispositivos, otro se creará utilizando el número que le sigue. Es decir, si abrimos el dispositivo */dev/bpf0*, inmediatamente el *kernel* creará el */dev/bpf1* para que otra aplicación pueda abrirlo y capturar/enviar paquetes.

  Además, antes de poder enviar un paquete, se deberá asociar el *file descriptor* asociado a ese dispositivo a una interfaz de red. Cuando se tiene la interfaz sobre la que se quiere mandar paquetes, se debe conseguir el índice asociada a ella (esto puede lograrse a través de una llamada a la función *if_nametoindex*). Luego, se debe utilizar la *system call ioctl*, utilizando el pedido *BIOCSETIF* para realizar esta asociación.

  Una vez que se realiza esta secuencia de operaciones, el *file descriptor* asociado a esa apertura podrá ser utilizado para enviar paquetes. Cada escritura sobre él implicará un envío por la interfaz asociada.

  Esto implica que, a diferencia de *GNU/Linux* en la que un mismo *socket* nos permitía enviar paquetes por diferentes interfaces de red, en *BSD* se tendrá que usar un descriptor por cada dispositivo de red por el que se quieran enviar paquetes que incluyan una trama de capa de enlace.

  La figura 26 muestra un ejemplo de este mecanismo.

Figura 26: Apertura de un socket de capa 2 y envío de un paquete en BSD

```
// La variable en la que se guardará el descriptor
int l2sock = -1;

// Iteramos desde i=0 hasta que podamos abrir el
// descriptor. Eventualmente, esto tiene que ocurrir.
for (int i = 0; sock == -1; i++) {
    // Creamos una cadena con formato "/dev/bpf"+i
    std::ostringstream oss;
    oss << "/dev/bpf" << i;

    // Intentamos abrir el dispositivo.
    // Si esta operación falla, va a retornar -1,
    // por lo que seguiremos iterando.
    sock = open(oss.str().c_str(), O_RDWR);
}

// Estructura para guardar los datos necesarios
// para asociar el descriptor a la interfaz.
struct ifreq ifr;
// Copiamos el nombre de la interfaz.
strncpy(
    ifr.ifr_name,
    NOMBRE_DE_INTERFAZ,
    sizeof(ifr.ifr_name) - 1
);

// Asociamos la interfaz al descriptor
if(ioctl(l2sock, BIOCSETIF, (caddr_t)&ifr) < 0) {
    close(l2sock);
    throw std::runtime_error(
        "Error al asociar el descriptor"
    );
}

// Enviamos un paquete
write(
    l2sock,
```

```
    BUFFER_PAQUETE,
    LONGITUD_PAQUETE
);
```

- La lectura de la tabla de rutas es un poco más compleja en *BSD*. Primero, se debe consultar la tabla al *kernel* a través de la *sysctl NET_RT_DUMP*. Esto va a devolver un bloque de *bytes* que internamente está compuesto por estructuras de tipo *rt_msghdr* que representan cada una de las entradas de la tabla.

  El mecanismo en sí es muy extenso como para incluirlo completo, pero la figura 27 muestra un resumen del algoritmo que se utilizó.

Figura 27: Lectura de la tabla de rutas en *BSD*

```
std::vector<char> query_route_table() {
    int mib[6] = {
        CTL_NET,
        AF_ROUTE,
        0,
        AF_INET,
        NET_RT_DUMP,
        0
    };
    std::vector<char> buf;
    size_t len;

    // Pedimos el tamaño de la tabla
    if (sysctl(mib, 6, NULL, &len, NULL, 0) < 0)
    {
        throw std::runtime_error("sysctl failed");
    }

    buf.resize(len);
    // Leemos la tabla
    if (sysctl(mib, 6, &buf[0], &len, NULL, 0) < 0)
    {
        throw std::runtime_error("sysctl failed");
    }
```

```
    return buf;
}

// Consultamos la tabla
std::vector<char> buffer = query_route_table();
char *next = &buffer[0],
     *end = &buffer[buffer.size()];

// Esta variable apuntará a cada entrada
rt_msghdr *rtm;
std::vector<sockaddr*> sa(RTAX_MAX);
char iface_name[IF_NAMESIZE];
while(next < end) {
    rtm = (rt_msghdr*)next;
    // Interpretar esta estructura
    parse_header(rtm, sa.begin());
    // Verificar que sea una entrada que especifique
    // una ruta
    if (sa[RTAX_DST] && sa[RTAX_GATEWAY] &&
        if_indextoname(rtm->rtm_index, iface_name))
    {
        // sa es una entrada en la ruta.
        // Se debería leer cada campo y completar
        // la estructura RouteEntry
    }
    next += rtm->rtm_msglen;
}
```

4.10.3 Windows

Para que estos mecanismos funcionen en *Windows*, hubo que hacer una implementación nueva, dado que son completamente incompatibles con los de *GNU/Linux* y *BSD*.

- En *Windows* directamente no se pueden enviar paquetes que incluyan una trama de capa de enlace. Esto fue implementado en el sistema operativo como una supuesta medida de seguridad, asumiendo que cualquiera que quisiese enviar un paquete manipulando la capa de enlace, estaría realizando acciones con fines maliciosos.

Existen artículos que explican formas de hacer esto, indicando que es necesario crear drivers adicionales e instalarlos en la máquina adonde se quieren enviar los paquetes. Como esto traería varias complicaciones y no se creía que el envío de paquetes de capa de enlace en *Windows* fuese un aspecto tan importante de la librería, se decidió no implementarlo. En un futuro podría tenerse en cuenta, pero se prefirió dirigir la atención a otros aspectos más importantes del proyecto.

- Leer la tabla de rutas es muy simple en *Windows*. Todo se reduce a utilizar a utilizar la función *GetIpForwardTable* de su *API* para leerla sobre una estructura de tipo *MIB_IPFORWARDTABLE*. Esta tabla tiene un arreglo con todas las entradas, por lo que sólo hay que iterarla y crear una *RouteEntry* para cada una.

  La figura 28 muestra en lineas generales al algoritmo que se usó.

Figura 28: Lectura de la tabla de rutas en *Windows*

```
// La estructura adonde se guardarán las entradas
MIB_IPFORWARDTABLE *table;
ULONG size = 0;

// Pedimos el tamaño de la tabla
GetIpForwardTable(0, &size, 0);
std::vector<uint8_t> buffer(size);
table = (MIB_IPFORWARDTABLE*)&buffer[0];

// Leemos la tabla
GetIpForwardTable(table, &size, 0);

// Iteramos todas las entradas
for (DWORD i = 0; i < table->dwNumEntries; i++) {
    MIB_IPFORWARDROW *row = &table->table[i];
    if(row->dwForwardType ==
        MIB_IPROUTE_TYPE_INDIRECT)
    {
        // Row apunta a una entrada.
        // Aca debería construirse un
        // objeto RouteEntry
    }
}
```

4.11 Portabilidad entre diferentes arquitecturas

Tal como se mencionó en secciones anteriores, *libtins* tenía que compilar y correr correctamente en diferentes arquitecturas. Se puso como objetivo a las más comunes, que pueden encontrarse tantos en computadoras hogareñas, como servidores y dispositivos embebidos, como *routers*.

Luego de buscar que tipos de arquitecturas se utilizan normalmente, se definió la siguiente lista que contiene todas a las que inicialmente se quiso dar soporte:

- *x86*: está basada en el set de instrucciones *Intel 8086* y es una de la más comunes en máquinas hogareñas en la actualidad. Es una arquitectura *little endian*.
- *x86-64*: es la versión de 64 *bits* de *x86*. Es también muy común en máquinas hogareñas. También utiliza la convención *litte-endian*.
- *ARM*: el uso de esta arquitectura se expandió muchísimo con el uso de dispositivos móbiles en los últimos años. Prácticamente todos los celulares que utilizan el sistema operativo *Android*, los *routers* que se encuentran en cualquier hogar y muchos más dispositivos embebidos la utilizan. Puede utilizar *big* o *little endian*.
- *MIPS*: es también ampliamente utilizada en dispositivos embebidos, como *routers*. También pueden utilizar tanto la convención *big* como *little endian*.

El mayor problema que se presentaba entre las diferentes arquitecturas era el uso de diferentes *endian*. Esto era problemático, dado que cada *PDU* utilizaba uno en particular, por lo que había que adaptar el código para que funcione en cualquier máquina, independientemente del *endian* que utilice.

Para lograr esto, se definieron diferentes funciones que convertían un valor de 16, 32 o 64 *bits* desde el *endian* que utilizara la máquina en la que se compilaba hacia *little* o *big endian* y viceversa. La figura 29 muestra una posible implementación de esta idea, aunque el código que se utilizó usa programación genérica utilizando *templates* para no definir tantas funciones cuyo cuerpo sea prácticamente el mismo.

Figura 29: Funciones de modificación de *endian*

```
// Convierte un valor de 16 bits del endian de
// la máquina a little endian
uint16_t host_to_le(uint16_t);

// Convierte un valor de 16 bits del endian de
// la máquina a big endian
uint16_t host_to_be(uint16_t);

// Convierte un valor de 16 bits de little endian
// al de la máquina
uint16_t le_to_host(uint16_t);

// Convierte un valor de 16 bits de big endian
// al de la máquina
uint16_t be_to_host(uint16_t);

// Lo mismo para 32 y 64 bits
// ....
```

Luego se utilizó compilación condicional para detectar en tiempo de compilación el *endian* utilizado y elegir que definición se usaría en cada función. Es decir, en una máquina *big endian*, la función *host_to_be* debería recibir un valor y retornarlo sin modificarlo, mientras que en *little endian* debería realizar la conversión apropiada.

Finalmente, se definieron métodos *getters* y *setters* en cada clase que representara a un *PDU* para pedir o asignar un valor a cada campo de su estructura interna. Cada *getter* convertiría el valor a retornar desde el *endian* que utilizara ese protocolo al de la máquina en la que se compilaba. El análogo ocurría para los métodos *setters*. Esa modificación podía no alterar el valor en absoluto si los dos *endian* (el del protocolo y el de la máquina) eran iguales, o podía modificarlo en caso que fuesen diferentes. La figura 30 muestra un ejemplo de la implementación de una clase que represente un *PDU* utilizando esta técnica.

Figura 30: Definición de campos *getters* y *setters*

```
// Una clase de ejemplo que representa al PDU de TCP.
// TCP utiliza big endian
class TCP {
public:
    // Setter del campo destination port
    void set_destination_port(uint16_t value) {
        // En caso de que la máquina sea big endian,
        // esto es la función identidad.
        // En caso de ser little endian, se modificará
        // el endian de "value"
        destination_port = host_to_be(value);
    }

    // Getter del campo destination port
    uint16_t get_destination_port() {
        // Si los endian difieren, modifica el valor,
        // en caso contrario lo retorna tal como
        // lo recibe.
        return be_to_host(destination_port);
    }
private:
    uint16_t destination_port;
};
```

De esta manera, los campos de cada clase siempre están en el *endian* que utiliza el protocolo. Al pedir o asignarle diferentes datos, se realiza la conversión. Esto es conveniente para que al momento de analizar/serializar las estructuras no haya que hacer ningún tipo de modificación a los valores. Se leen, se escriben y se guardan en memoria en el mismo *endian*.

Una vez que se implementaron estos mecanismos de conversión de *endian* se comenzó a probar los diferentes *tests* en las diferentes arquitecturas.

4.11.1 *x86*

Inicialmente se comenzó desarrollando la librería en una máquina que utilizaba *x86*. Como consecuencia, no hubo nada que portar. Las pruebas implementadas indicaban que todo funcionaba bien en esta arquitectura.

4.11.2 *x86-64*

Dado que ya funcionaba en máquinas *x86*, y las dos arquitecturas usan el mismo *endian*, no se debería tener que cambiar nada para que funcione en *x86-64*. Sólo se debe tener en cuenta que la longitud en *bytes* de algunos tipos de datos estándar puede cambiar entre las dos arquitecturas.

Esto no fue un problema porque para todas las estructuras pertenecientes a los diferentes *PDU*, en las que se querían campos de una longitud específica de *bits*, se utilizaron los tipos de datos que provee la librería estándar de *C++*, como *uint64_t* o *uint32_t*, para indicar exactamente de cuantos debía ser cada campo. De esta manera, hacer que la librería funcione en *x86-64* no trajo ningún tipo de inconveniente.

4.11.3 *ARM*

Para portar a esta arquitectura, como para cualquier otra, era necesario probar que el código corriese bien en ella. Para hacer esto, se podían encarar dos caminos:

- Instalar una máquina virtual con arquitectura *ARM* y correr los tests ahí dentro. Previamente ya se había creado una máquina que utilizara *ARM*, pero se notó que la misma funcionaba extremadamente lento, dado que la máquina *host* tenía que emular la arquitectura.

- Utilizar un *cross-compilador*. Los *cross-compiladores* permiten compilar código desde una arquitectura *X* para que el binario resultante pueda ser utilizado en otra arquitectura *Y*. De esta manera, se podría hacer todo sobre la misma máquina física, luego *cross-compilar* y ejecutar el código utilizando las utilidades de *qemu*, que permiten correr un binario de otra arquitectura sin necesidad de instalar todo un sistema operativo.

Se optó por la utilización de un *cross-compilador* debido a su facilidad de uso. Además, ya existen paquetes en los repositorios de código que contienen versiones de *gcc* que compilan código para *ARM*.

Como consecuencia, simplemente hubo que compilar la librería y los *tests* utilizando el *cross-compilador* y comenzar a arreglar todo lo que no funcionara correctamente. Tuvieron que modificarse las definiciones de varios *PDU* utilizadas en arquitecturas *big endian* porque no eran correctas.

4.11.4 *MIPS*

Una vez que ya todo funcionaba en *ARM*, se probó en *MIPS*. Como ya todos las pruebas pasaban en *big endian*, no hubo que hacer ninguna modificación en el código.

5 Capítulo 5: *Benchmarks*

Luego de desarrollar la librería y dejarla en un estado estable, se procedió a realizar *benchmarks* para comparar su *performance* con la de otras que realizaran la misma tarea.

Un objetivo de los *benchmarks* que se crearon fue que sean reproducibles por quien quisiese hacerlo. Como consecuencia, no era posible realizar capturas grandes de tráfico y luego utilizarlas para ver como se comportaban las diferentes librerías, dado que harían falta utilizar archivos bastante grandes y luego se debería utilizar algún servidor para alojarlos, lo que implicaba trabajo/costos de más.

Fue por ésto que se terminó optando por el siguiente esquema para las pruebas:

- Primero, se correría una aplicación que genere un archivo *pcap* que contenga una gran cantidad de cierto tipo de paquetes.
- Ejecutar un test para cada librería a probar sobre el archivo generado en el punto anterior.
- Generar un archivo con los resultados, indicando cuanto tardó cada una en procesarlo.
- Correr algun *script* que procesara los resultados y generara diferentes visualizaciones, incluyendo tablas en *HTML* y gráficos.

De esta manera, cualquiera podría correr los *benchmarks* en su PC, siempre y cuando tenga instaladas las librerías a probar.

La prueba que se utilizó para cada librería constaba de la apertura del archivo de entrada y la interpretación de cada paquete en él, utilizando los mecanismos provistos por cada una de ellas. Cada prueba utilizaría algún mecanismo del lenguaje para medir el tiempo que se tardaba en interpretar todos ellos, sin tener en cuenta el tiempo de apertura/cierre del archivo de entrada. Además, estas repiten el mecanismo 3 veces, y utilizarían el promedio del tiempo de cada una de ellas.

Finalmente, la ejecución de los *benchmarks* generará un archivo que incluye el tiempo que tardó cada librería para cada test.

5.1 Librerías a probar

Al desarrollar las pruebas, se buscaron las librerías más utilizadas para realizar análisis de paquetes. Únicamente se tuvieron en cuenta aquellas que

estuviesen implementadas en los lenguajes *C*, *C++*, o *Python* para reducir la cantidad a probar y utilizar tecnologías que se conociesen de antemano. En un futuro se podrían extender las pruebas a otros lenguajes, pero por ahora esto es suficiente.

Luego de investigar cuáles eran las más utilizadas, se terminaron eligiendo las siguientes:

- *scapy*[3]: desarrollada en *Python*, es una librería y a su vez un herramienta de análisis y construcción/envío de paquetes muy difundida. Es muy flexible y simple de utilizar, permitiendo leer y modificar paquetes con apenas unas lineas de código.

- *dpkt*[27]: otra implementada en *Python*. Es también muy simple de utilizar, y además se la promociona como una librería rápida, por lo que sin dudas se incluyó en la lista.

- *impacket*[26]: también desarrollada en *Python*. Tiene soporte para varios protocolos de capa de aplicación y está orientada a realizar tareas relacionadas con seguridad informática.

- *libcrafter*[28]: una librería que fue creada al mismo tiempo que se desarrollaba *libtins*, también implementada en *C++*. Se la tuvo en cuenta dado que utilizaba el mismo lenguaje y podría ser un buen punto de comparación.

- *libpcap*[29]: esta es la librería desarrollada en el lenguaje *C* es la más utilizada para realizar captura de paquetes, sin proveer mecanismos para su análisis e interpretación. Esta se utilizó únicamente para tener un punto de comparación al probar las otras. Ninguna librería que implemente análisis e interpretación de paquetes podría funcionar más rapido que *libpcap*, que sólo permite capturarlos.

5.2 Características del ambiente de pruebas

Todas las pruebas se corrieron sobre la misma máquina y sistema operativo. Las características de la misma fueron:

- **CPU**: Intel Core i7-2670QM (2.20Hz).

- **Memoria**: 8GB DDR3 RAM (1333 MHz).

- **Sistema operativo**: Linux mint 17 64bits, kernel 3.13.0.

- **Compilador**: gcc 4.8.2.

Por otro lado, se utilizaron las siguientes librerías:

- *libtins* compilada usando el parámetro *–enable-c++11* de configuración..
- *libcrafter* versión 0,3.
- *scapy* versión 2,2.
- *impacket* versión 0,9,11.
- *dpkt* versión 1,8.
- *libpcap* versión 1,5,3.

5.3 Código fuente

Se creó un repositorio para los *benchmarks* en *github*. De esta manera, cualquiera puede clonarlo, compilar las pruebas y correrlas en cualquier entorno. Éste está disponible en la siguiente *URL*:

`https://github.com/mfontanini/packet-capture-benchmarks`

El repositorio incluye un archivo con instrucciones para poder compilarlo. Siempre y cuando se tengan instaladas las librerías a probar en lugares estándar, se reduce a utilizar la secuencia clásica de comandos de compilación, tal como lo muestra la figura 31.

Figura 31: Compilación de los benchmarks

```
# Generamos el archivo Makefile
./configure
# Compilamos el proyecto
make
```

Se incluyeron opciones de configuración del proyecto para deshabilitar la prueba de cada librería, de manera que se pueda elegir qué *tests* correr. También se agregaron otras para indicar la ubicación de los archivos necesarios para compilarlos, en caso de que no estén en lugares estándar.

La ejecución de los *benchmarks* puede hacerse directamente a través del comando *make*, tal como lo muestra la figura 32.

Figura 32: Ejecución de los benchmarks

```
# Ejecutamos los benchmarks
make benchmark
```

Esto genera un archivo de texto plano que contiene los resultados. Se incluye una utilidad implementada en *Python* que procesa éstos y genera salidas en diferentes formatos, como tablas o gráficos en *HTML*.

Figura 33: Ejecución de la utilidad que procesa los resultados

```
# Toma la entrada de archivo_entrada
# Genera un gráfico HTML sobre archivo_salida.html
./results_analyzer.py archivo_entrada \
    archivo_salida.html html-graphic
```

5.4 Resultados generados

En las siguientes secciones se mostrarán los resultados de las pruebas luego de correrlas en el ambiente mostrado en la sección 5.2. Los mismos serán mostrados de dos formas diferentes:

1. Una tabla que muestra para cada librería cuanto tiempo tardó en procesar la entrada utilizada. Además, se agregó una columna que indica cuantos paquetes por segundo podría procesar siguiendo esa velocidad de procesamiento.

2. Un gráfico con los tiempos de cada una, para hacer más simple su comparación. Para esta visualización de los resultados no se tuvo en cuenta la librería *scapy*, dado que tarda tanto en procesar los datos que si se incluyera, la altura del resto de las barras no se distinguiría correctamente.

5.5 Primer *benchmark*: *TCP* + datos

El primer *benchmark* constó de generar un archivo con 500,000 paquetes iguales, formados por:

- Una trama *Ethernet*.
- Un paquete *IP*.

- Un segmento *TCP*.
- Un *payload* de capa de aplicación de 742 *bytes*.

Con esto estaríamos probando la capacidad de analizar paquetes que podría verse en la red al transferir datos entre 2 maquinas. El tamaño del *payload* utilizado se eligió de manera que el archivo sobre el que se generara la entrada de las pruebas no terminase siendo demasiado grande.

5.5.1 Resultados

Los resultados obtenidos pueden verse en la tabla 1.

Librería	**Tiempo tomado (en segundos)**	**Paquetes por segundo**
libpcap	0.141	3.546.099
libtins	0.312	1.602.564
dpkt	8.132	61.485
libcrafter	12.209	40.953
impacket	18.5	27.027
scapy	187.082	2.672

Cuadro 1: Resultados del *benchmark* 1

En la figura 34 puede verse un gráfico de barras mostrando los resultados encontrados.

Como puede verse, *libtins* no sólo tarda muy poco más que *libpcap*, sino que es mucho más rapida que el resto de las librerías probadas.

5.6 Segundo *benchmark*: *TCP* + *opciones* + datos

En el segundo *benchmark* también se generaron 500,000 paquetes iguales, pero se agregaron opciones a los segmentos *TCP*. Por lo tanto, cada paquete se veía de la siguiente forma:

- Una trama *Ethernet*.
- Un paquete *IP*.
- Un segmento *TCP* que contenía.
  - Una opción de tipo *máximum segment size*;
  - Una opción de tipo *window scale*.

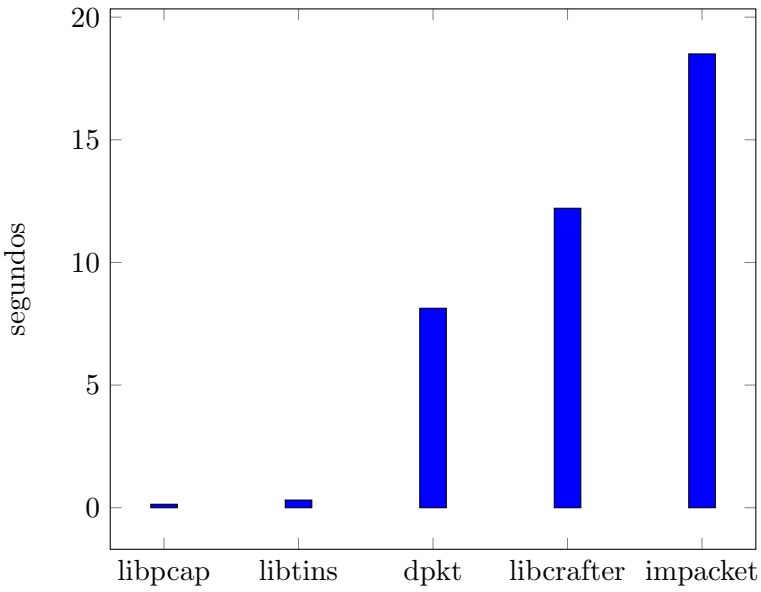

Figura 34: *Gráfico mostrando los resultados del primer benchmark*

  - Una opción de tipo *SACK-permitted*.
  - Una opción de tipo *selective ACK*.
  - Una opción de tipo *timestamp*.
  - Una opción de tipo *alternate checksum*.
- Un *payload* de capa de aplicación de 742 *bytes*.

Estos son obviamente poco realistas, dado que nunca se verían tantas opciones *TCP* juntas en tantos paquetes seguidos. Pero sirve como buen punto de comparación para ver cuan rápido puede analizar las opciones cada librería.

5.6.1 Resultados

Los resultados pueden verse en la tabla 2 y el gráfico 35.

Librería	Tiempo tomado (en segundos)	Paquetes por segundo
libpcap	0.145	3.448.275
libtins	0.48	1.041.666
dpkt	8.508	58.768
libcrafter	20.199	24.753
impacket	44.491	11.238
scapy	194.543	2.570

Cuadro 2: Resultados del *benchmark* 2

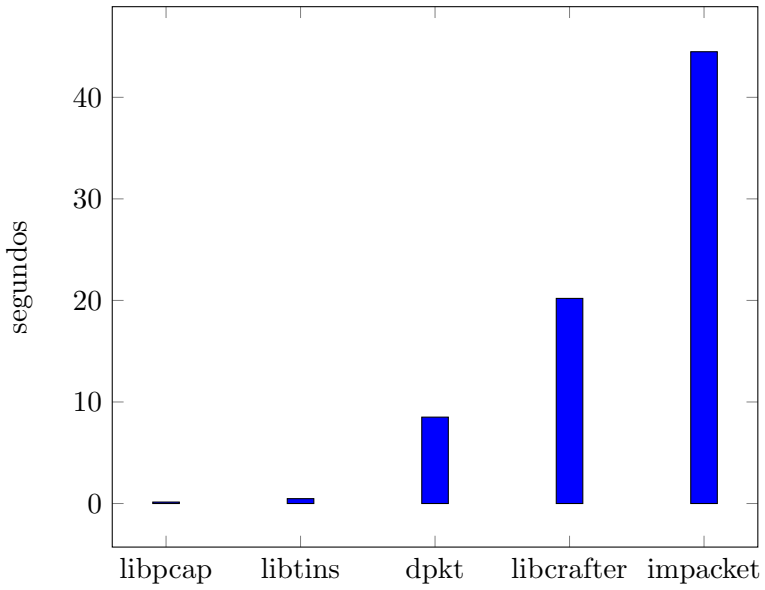

Figura 35: *Gráfico mostrando los resultados del segundo benchmark*

En este caso puede verse que *libtins* tuvo un *overhead* mayor que las demás librerías, en comparación con los resultados *benchmark* anterior. Pero de todas formas, sigue siendo aproximadamente 17 veces más rápida que *dpkt*.

5.7 Tercer *benchmark*: respuestas *DNS*

Este último *benchmark* probó el comportamiento al analizar paquetes *DNS*. Se crearon 500,000 paquetes *DNS*, cada uno con la siguiente estructura.

- Una trama *Ethernet*.

- Un paquete *IP*.
- Un segmento *UDP*.
- Un *payload* de capa de aplicación de 46 *bytes*, cuyo contenido eran datos de una respuesta *DNS* con las siguientes propiedades relevantes:
  - Una entrada en la sección de consultas, pidiendo el registro de tipo *NS*, clase *IN* del dominio *google.com*.
  - Un registro en la sección de respuesta, conteniendo el dominio *ns4.google.com*.

Los paquetes *DNS* necesitan un análisis más exhaustivo que todos los paquetes probados anteriormente. Esto se debe a que usan un tipo de codificación y compresión que reduce la cantidad de *bytes* que se necesitan enviar y recibir para resolver dominios. Esto daría una buena medida de la velocidad que tarda cada librería en procesar los paquetes de este protocolo.

5.7.1 Resultados

La tabla 3 y el gráfico 36 muestran los resultados de esta prueba

Librería	**Tiempo tomado (en segundos)**	**Paquetes por segundo**
libpcap	0.036	13.888.888
libtins	0.377	1.326.259
libcrafter	13.524	36.971
dpkt	23.319	21.441
impacket	36.487	13.703
scapy	374.637	1334

Cuadro 3: Resultados del *benchmark* 3

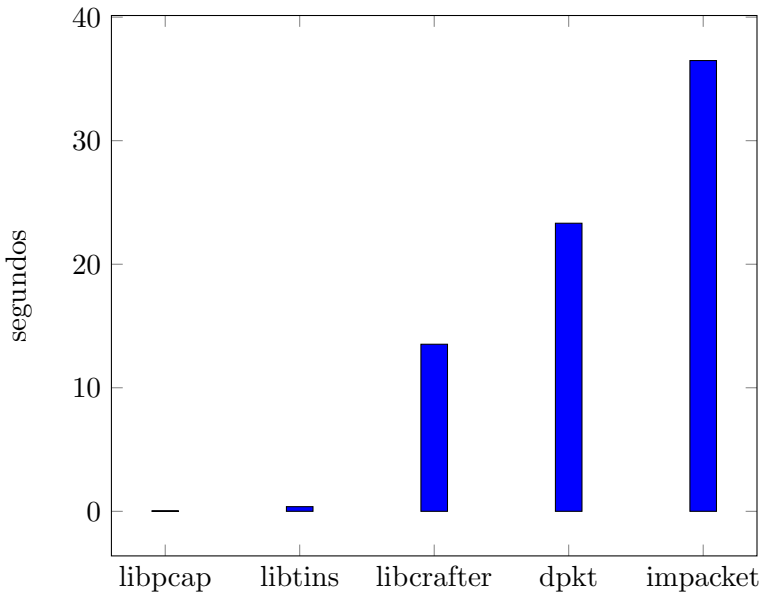

Figura 36: *Gráfico mostrando los resultados del tercer benchmark*

Nuevamente puede verse que *libtins* es mucho más rápida que el resto, a pesar de que agrega un *overhead* bastante grande a lo que tarda *libpcap*. Sin embargo, dado que los paquetes *DNS* implican una cierta complejidad de análisis, estos resultados fueron más que aceptables.

5.8 Conclusiones

Como puede verse, todos los *benchmarks* realizados comprueban que *libtins* interpreta paquetes por lo menos 10 veces más rápido que cualquier otra librería probada. Además, se puede ver que el *overhead* que agrega en comparación con *libpcap* es bastante bajo.

6 Capítulo 6: Ejemplos y casos de uso

En esta sección se introducen diferentes casos de uso en los que se podría usar *libtins*. En el sitio web de la librería, se incluyen varios ejemplos[30] cortos que muestran la utilidad y facilidad con la que se pueden desarrollar herramientas que monitoreen y generen tráfico.

Muchas de estas aplicaciones necesitarían tener acceso a todo el tráfico de la red a analizar para funcionar correctamente. Esto podría lograrse configurando un puerto espejo en algún *switch* por el que pase el tráfico o utilizar algún mecanismo de captura remoto.

En todos los casos de uso planteados, *libtins* sería de gran utilidad para poder acceder a los datos de cada capa, incluidos los datos de la de aplicación. Además, funcionaría independientemente de los protocolos utilizados en capas inferiores. Por ejemplo, no habría que modificar un código para acceder a los datos de cada de transporte de un paquete si éste estuviese encapsulado con *IEEE 802.1q* (protocolo utilizado al usar *VLAN*s), si simplemente utilizara *Ethernet* como protocolo de capa de enlace o si fuese un paquete *IPv6*.

6.1 Detección de tráfico malicioso

Un escenario en el que la librería podría ser utilizada es para la realización de monitoreo de redes con el objetivo de detectar tráfico malicioso. En las siguientes secciones se indicarán algunos de los eventos específicos que se podrían buscar.

6.1.1 Explotación de vulnerabilidades

Ciertos tipos de ataques, principalmente los que tienen como objetivo servidores *Web*, pueden ser detectados analizando el contenido de los datos que envía el cliente y buscando ciertos patrones en él. En el caso de *HTTP*, esto incluiría ataques como *SQL Injection*[32] y *Code injection*[33], que generalmente se realizan a través del envío de datos que suelen caer dentro de ciertos formatos que son fáciles de detectar.

Para lograr detectar estos patrones correctamente, sería necesario reconstruir los flujos *TCP* que envían los clientes. Esto se puede lograr haciendo un seguimiento de los números de secuencia de los paquetes y almacenando los datos que se incluyen en ellos hasta poder reconstruir los que originalmente se enviaron. Esto implicaría utilizar almacenamiento temporario para guardar los paquetes que llegan fuera de orden hasta que éstos puedan reordenarse. Una vez que se obtienen los datos que envió el cliente, se

puede aplicar un análisis para extraer datos específicos del protocolo que se esté usando, y luego aplicar una serie de expresiones regulares que intenten buscar los patrones mencionados con anterioridad.

Esto podría detectarse instalando aplicaciones o módulos adicionales en cada servidor, como *ModSecurity*[31], pero implicaría que cada vez que se cree uno nuevo, se los deba instalar para mantenerlo seguro. Al crear una aplicación que monitoree todo el tráfico de la red buscando este tipo de tráfico, no es necesario realizar ninguna modificación al instalar nuevos servidores, simplemente que el tráfico que envía y recibe pueda ser capturado por el monitor.

Existen varias aplicaciones que cumplen este objetivo, como *Snort*[34] y *Bro*[35].

6.1.2 Ataques de fuerza bruta

Es muy común que atacantes realicen ataques de fuerza bruta sobre los servicios públicos de una red para intentar acceder a recursos internos. Esto implica hacer conexiones a servidores e intentar autenticarse repetidas veces utilizando diferentes credenciales hasta lograr una autenticación exitosa. El mecanismo de autenticación va a variar dependiendo del protocolo sobre el que se este realizando el ataque. Estos normalmente pueden ser:

- *SSH* (*Secure Shell*): este protocolo permite ejecución remota de comandos sobre servidores, así como también transferencia de archivos y otras acciones administrativas. Acceder a un servicio como éste le permitiría a un atacante ejecutar aplicaciones dentro del servidor al que acceda, lo que lo habilitaría utilizar el servidor para diferentes fines maliciosos.

- Servidores de base de datos: esto incluiría a diferentes *DBMS* (*Database Management System*) como *MySQL*[36], *SQL Server*[37] y *PostgreSQL*[38], entre otros. Si un atacante lograra autenticarse exitosamente a un servidor de este tipo, podría acceder a valiosa información interna.

- Servidores *HTTP*: esto involucraría tanto autenticación contra diferentes aplicaciones *Web*, así como también directamente por *HTTP*, utilizando los diferentes mecanismos que provee el protocolo.

- Servidores de correo electrónico: si un atacante pudiese autenticarse contra un servidor de correo electrónico, este podría ser utilizado para enviar correo basura o *SPAM*.

Estos ataques pueden detectarse de diferentes maneras:

- Viendo la cantidad de conexiones que realiza un mismo cliente sobre uno o más servidores en un período de tiempo corto. Esto no es útil para detección de fuerza bruta sobre servidores *HTTP*, dado que es muy normal ver múltiples conexiones simultáneas hacia éstos, pero sí sirven en muchos otros. Si un usuario realiza demasiadas conexiones, se podría generar una alerta para que un administrador verifique qué es lo que está haciendo.

- Analizar los datos que envía el cliente para detectar intentos de inicio de sesión. Esto no puede realizarse en todos los protocolos, dado que muchos de ellos utilizan cifrado. Sin embargo, en algunos podría permitir incluso observar las credenciales que se envían, lo que facilitaría la detección del ataque.

6.2 Monitoreo de redes

Otra aplicación de la librería es el monitoreo de redes. En este caso, más que detectar tráfico malicioso, se buscaría extraer información sobre:

- Conexiones realizadas por usuarios de la red: con esto se podría controlar a qué sitios se conectan e incluso generar alertas cuando se visitan algunos en particular que no deberían visitarse.

- Conexiones realizadas hacia servidores de la red: esto permitiría ver en tiempo real qué servidores están siendo accedidos, y desde qué direcciones *IP*.

- Detección de problemas en la red: se podrían encontrar diferentes métricas que permitan encontrar problemas en la red o en alguna ruta en particular. Las métricas podría ser cantidad de paquetes perdidos, tiempo de respuesta de los servidores, etc.

Para implementar aplicaciones de este tipo se tendría que:

- Seguir y reconstrucción de flujos *TCP*: al igual que en la detección de tráfico malicioso, es necesario reconstruir los flujos *TCP* enviados tanto por el cliente como por el servidor para poder extraer datos de la capa de aplicación. Además, al seguir los diferentes flujos se podrían encontrar cortes abruptos en las conexiones, o fallas durante el establecimiento de una conexión, entre otras cosas.

- Analizar pedidos y respuestas *DNS*. Observar tráfico *DNS* permite saber el nombre de dominio de los servidores a los que se están conectando los clientes. De otra manera, sólo se tendría acceso a su dirección *IP*, lo que es poco informativo desde un punto de vista administrativo.

- Análisis de capa de aplicación. Al observar los datos de capa de aplicación se pueden extraer más información que facilite saber qué es lo que esta haciendo un cliente al conectarse a algún servidor. Por ejemplo, si se analizara el tráfico *HTTP*, se podría saber a qué *URL*s está accediendo el usuario.

7 Capítulo 7: Conclusiones y trabajo futuro

7.1 Conclusiones

La implementación desarrollada en esta tesina terminó cumpliendo con las expectativas propuestas antes de comenzarla. Para poder realizar el desarrollo correctamente hubo que investigar sobre las definiciones de cada protocolo y como se enlazaban las diferentes capas de un paquete entre sí. Además, hubo que buscar información acerca de como se manejan muchos mecanismos internos que utilizan los diferentes sistemas operativos para manejar interfaces de red, enviar y capturar paquetes.

La librería tuvo una recepción muy buena por parte de personas externas al proyecto. Actualmente, hay por lo menos un nuevo mensaje en el foro del proyecto por semana, por parte de gente que tiene consultas o reportes de *bugs*. Hay varias personas e incluso algunas empresas privadas que optaron por usar *libtins* para sus desarrollos.

El proyecto cumplió con las expectativas de eficiencia que se plantearon, que eran de gran importancia para su utilidad. Este es uno de los motivos por los que muchos podrían optar por usarla: esta implica un costo muy bajo a pagar en términos de poder de procesamiento, y que como consecuencia permite procesar paquetes de forma muy simple y sin tener que lidiar con aspectos internos de cada protocolo.

La combinación de alguna aplicación de captura de paquetes, como puede ser *tcpdump*[39], junto con *libtins* para realizar un procesamiento fuera de linea de los mismos sería de gran utilidad para ciertos tipos de aplicaciones o entornos en los que no se tenga un poder de procesamiento lo suficientemente alto como para realizarlo en tiempo real. Además, la posibilidad de utilizar la librería en entornos embebidos, como *routers*, expande sus casos de uso enormemente.

7.2 Trabajo futuro

El desarrollo de la librería sigue muy activo. Si bien cumplió con los objetivos que se plantearon inicialmente, su desarrollo no llegó a su fin. Con regularidad se agregan o implementan mecanismos internos de una forma más eficiente.

Como trabajo futuro se podría plantear la implementación de más protocolos, de manera que cualquiera que quiera usarla, sin importar el objetivo que tenga, pueda usarla sin tener que toparse con que necesita usar protocolos para los que no se tiene soporte.

Además, se podrían agregar diferentes utilidades que hagan tareas específicas con los diferentes paquetes. Por ejemplo, se podría implementar una clase que realice reordenamiento de paquetes *TCP* para poder procesar el contenido de un flujo de forma ordenada.

Referencias

[1] *Wireshark*, https://www.wireshark.org/

[2] *Python*, https://www.python.org/

[3] *Scapy*, herramienta interactiva de manipulación de paquetes, SecDev, http://www.secdev.org/projects/scapy/

[4] *libtins*, librería de captura y creación de paquetes, https://libtins.github.io/

[5] W. Richard Stevens, *TCP/IP Illustrated, Vol. 1*, Addison-Wesley, Segunda edición, 2011

[6] Charles M. Kozierok, *The TCP/IP Guide: A Comprehensive, Illustrated Internet Protocols Reference*, No Starch Press, Primera edición, 2005

[7] *IEEE 802.3: Ethernet*, http://standards.ieee.org/about/get/802/802.3.html

[8] *RFC 1661: Point-to-Point Protocol*, http://www.ietf.org/rfc/rfc1661.txt

[9] *IEEE 802.11: Wireless LANs*, http://standards.ieee.org/about/get/802/802.11.html

[10] *RadioTap*, http://www.radiotap.org/

[11] *RFC 791: Internet Protocol*, http://www.ietf.org/rfc/rfc791.txt

[12] *RFC 2460: Internet Protocol, Version 6*, http://www.ietf.org/rfc/rfc2460.txt

[13] *RFC 793: Transmission Control Protocol*, http://www.ietf.org/rfc/rfc793.txt

[14] *RFC 768: User Datagram Protocol*, http://www.ietf.org/rfc/rfc768.txt

[15] *RFC 4960: Stram Control Transmission Protocol*, http://tools.ietf.org/html/rfc4960

[16] *RFC 2616: Hypertext Transfer Protocol – HTTP/1.1*, http://tools.ietf.org/html/rfc2616

[17] *RFC 1035: Domain Names - Implementation and Specification*, http://www.ietf.org/rfc/rfc1035.txt

[18] *RFC 2131: Dynamic Host Configuration Protocol*, http://www.ietf.org/rfc/rfc2131.txt

[19] *Endianness*, http://en.wikipedia.org/wiki/Endianness

[20] *Request For Comments*, http://www.ietf.org/rfc.html

[21] *Internet Engineering Task Force*, http://www.ietf.org/

[22] *HyperText Markup Language*, http://www.w3.org/html/wg/drafts/html/master/

[23] *Extensible Markup Language*, http://www.w3.org/TR/REC-xml/

[24] *GNU Lesser General Public License*, https://www.gnu.org/licenses/lgpl.html

[25] *The BSD 2-Clause License*, http://opensource.org/licenses/BSD-2-Clause

[26] *impacket*, librería de creación y decodificación de paquetes, coresecurity, https://code.google.com/p/impacket/

[27] *dpkt*, librería de creación y decodificación de paquetes, https://code.google.com/p/dpkt/

[28] *libcrafter*, librería de alto nivel para la creación y decodificación de paquetes de red, Esteban Pellegrino, https://github.com/pellegre/libcrafter

[29] *libpcap*, librería de monitoreo de redes a bajo nivel, http://www.tcpdump.org/

[30] *Ejemplos del uso de libtins*, http://libtins.github.io/examples/

[31] *ModSecurity*, Módulo de cortafuegos *Web* para el servidor *HTTP Apache*. https://www.modsecurity.org/

[32] *SQL Injection*, https://www.owasp.org/index.php/SQL_Injection

[33] *Code injection*, https://www.owasp.org/index.php/Code_Injection

[34] *Snort*, Sistema de prevención de intrusos, https://www.snort.org/

[35] *Bro*, Monitor de seguridad en redes, http://www.bro.org/

[36] *MySQL*, Servidor de base de datos *open source*, http://www.mysql.com/

[37] *SQL Server*, Servidor de base de datos, http://www.microsoft.com/en-us/server-cloud/products/sql-server/

[38] *PostgresSQL*, Servidor de base de datos *open source*, http://www.postgresql.org/

[39] *tcpdump*, Aplicación de captura de paquetes *open source*, http://www.tcpdump.org/