Universidad Zaragoza
1542

Trabajo Fin de Máster

Optimización del rendimiento en redes SIP mediante la aplicación de la tecnología eBPF en entornos de telecomunicaciones

Performance optimization in SIP networks through the application of eBPF technology in telecommunications environments

Autor
Carlos Irigoyen Álvarez

Director
Dayana Ribas González

Ponente
Álvaro Alesanco Iglesias

Escuela de Ingeniería y Arquitectura
2025

Resumen

En el ámbito de las telecomunicaciones, especialmente en el tráfico de telefonía IP, el rendimiento y la eficiencia en la gestión del tráfico son factores clave para garantizar una experiencia de usuario satisfactoria. En este contexto, el protocolo SIP (por sus siglas en inglés: Session Initiation Protocol) se encarga de establecer, mantener y finalizar las llamadas telefónicas en redes VoIP. Para las empresas operadoras, resulta esencial optimizar estos procesos, no solo para garantizar un servicio de calidad, sino también para obtener estadísticas precisas y mejorar la gestión de sus sistemas. Esto lleva a la necesidad de investigar y aplicar tecnologías avanzadas que permitan una mayor eficiencia en el tratamiento del tráfico de red.

Este Trabajo Fin de Máster se centra en la implementación y análisis de un firewall basado en tecnología eBPF (por sus siglas en inglés: *Extended Berkeley Packet Filter*) junto con XDP (por sus siglas en inglés: Express Data Path) en entornos Linux, con el objetivo de aportar ventajas en términos de rendimiento y capacidad de análisis del tráfico SIP. La empresa colaboradora en este proyecto, BTS, cuenta con una amplia experiencia en soluciones de VoIP, lo que refuerza la aplicación práctica de este estudio en entornos reales.

Para desarrollar el proyecto, se empleará Proxmox, una plataforma de virtualización utilizada por BTS en la mayoría de sus servidores, y que ofrece múltiples posibilidades para la creación de entornos de pruebas virtualizados. El trabajo se divide en varias fases: en primer lugar, el estudio de la tecnología eBPF y sus ventajas frente a soluciones tradicionales; en segundo lugar, la implementación del firewall en un entorno de pruebas basado en máquinas virtuales; y, finalmente, la realización de pruebas de rendimiento y análisis comparativo. Adicionalmente, como parte del proyecto se desarrollará una plataforma web centralizada para gestionar de forma sencilla y eficiente los firewalls eBPF desplegados en diferentes máquinas de la red. Este sistema permitirá a los trabajadores configurar, supervisar y administrar las reglas del firewall desde un único punto, gracias a una interfaz intuitiva diseñada para mejorar la usabilidad.

El objetivo final del proyecto es evaluar el impacto de esta tecnología en entornos de tráfico telefónico, determinando si su implementación podría aportar beneficios significativos en términos de rendimiento, escalabilidad y facilidad de gestión frente a las herramientas actualmente utilizadas por la empresa. Este análisis permitirá extraer conclusiones aplicables a la mejora de los sistemas actuales, optimizando el procesamiento de tráfico SIP en redes VoIP y facilitando la gestión centralizada de los firewalls.

Abstract

In the field of telecommunications, especially in IP telephony traffic, performance and efficiency in traffic management are key factors in ensuring a satisfactory user experience. In this context, the SIP protocol is responsible for establishing, maintaining and terminating telephone calls in VoIP networks. For carriers, it is essential to optimize these processes, not only to ensure quality service, but also to obtain accurate statistics and improve the management of their systems. This leads to the need to research and apply advanced technologies that allow greater efficiency in the treatment of network traffic.

This Master Thesis focuses on the implementation and analysis of a firewall based on eBPF (Extended Berkeley Packet Filter) technology together with XDP (Express Data Path) in Linux environments, with the aim of providing advantages in terms of performance and analysis capacity of SIP traffic. The collaborating company in this project, BTS – Business Telecommunications Services, has extensive experience in VoIP solutions, which reinforces the practical application of this study in real environments.

The project will be developed using Proxmox, a virtualization platform used by BTS in most of its servers, which offers multiple possibilities for the creation of virtualized test environments. The work is divided into several phases: firstly, the study of eBPF technology and its advantages over traditional solutions; secondly, the implementation of the firewall in a test environment based on virtual machines; and finally, performance testing and benchmarking. Additionally, as part of the project, a centralized web platform will be developed to easily and efficiently manage the eBPF firewalls deployed in different machines of the network. This system will allow workers to configure, monitor and manage firewall rules from a single point, thanks to an intuitive interface designed to improve usability.

The ultimate goal of the project is to evaluate the impact of this technology in telephone traffic environments, determining whether its implementation could bring significant benefits in terms of performance, scalability and ease of management compared to the tools currently used by the company. This analysis will allow to draw conclusions applicable to the improvement of current systems, optimizing the processing of SIP traffic in VoIP networks and facilitating the centralized management of firewalls.

Índice

Capítulo 1: Introducción 7
1.1. Introducción 7
1.2. Motivación y objetivos 9
1.3. Organización de la memoria 10
Capítulo 2: Análisis previo 11
2.1. Protocolos, herramientas y lenguajes utilizados 11
2.2. Estudio y análisis de la tecnología eBPF 18
2.2.1. Historia y fundamentos de eBPF 18
2.2.2. Uso actual en la industria 19
2.3. Sistema actual y planteamiento del problema 20
2.4. Propuesta de solución: Firewall Linux con eBPF 24
Capítulo 3: Exposición del proyecto a desarrollar 28
3.1. Descripción y bases del proyecto 28
3.2. Entorno de trabajo 29
3.3. Tecnologías aplicadas 32
3.4. Sistema completo 35
3.5. Instalación y configuración de herramientas a utilizar 37
Capítulo 4: Desarrollo técnico completo del Firewall Linux 39
4.1. Programa eBPF para el análisis de paquetes 40
4.2. Programa backend para los agentes: “add_rule.py” 44
4.3. Programa backend para el eBPF Master: “app.py” 47
4.4. Frontend en el eBPF Master: Web de gestión centralizada 49
4.5. Aplicación de políticas de seguridad 53
4.6. Puesta en marcha del sistema 53
Capítulo 5: Pruebas de funcionamiento y resultados 54
5.1. Pruebas iniciales 54
5.2. Pruebas realistas con tráfico SIP 54
5.2.1. Escenario 1: Carga media 56
5.2.2. Escenario 2: Carga alta no prolongada 58
5.2.3. Escenario 3: Carga alta prolongada 60
Capítulo 6: Conclusiones y líneas futuras 63
6.1. Conclusiones 63

6.2. Líneas futuras 64
Capítulo 7: Anexos 65
ANEXO A. Protocolo SIP. 65
ANEXO B. Proyectos, patentes y empresas que utilizan eBPF. 66
ANEXO C. PostgreSQL - Gestión y comandos. 68
ANEXO D. Proceso de instalación de herramientas utilizadas 69
ANEXO E. Desarrollo del código. 72
ANEXO F. Aplicación de políticas de seguridad y puesta en marcha del sistema. 87
ANEXO G. Pruebas de funcionamiento generales. 90
Funcionalidades del Firewall 90
Pruebas básicas iniciales con tráfico 91
ANEXO H. Complementos a las pruebas de carga. 94
ANEXO I. Uso de la herramienta SIPp. 99
Bibliografía 102
Glosario de términos y siglas. 103

Capítulo 1: Introducción

1.1. Introducción

Hoy en día, las tecnologías de comunicación están presentes en todos los aspectos de nuestras vidas y no dejan de evolucionar. Una de las tecnologías que ha revolucionado la comunicación empresarial es la telefonía IP (VoIP). Gracias a ella, las empresas y usuarios pueden realizar videoconferencias, compartir archivos y datos, y mantener una comunicación constante y efectiva en tiempo real.

La telefonía IP presenta múltiples ventajas frente a la telefonía convencional: costes reducidos, mayor flexibilidad y escalabilidad, así como funciones avanzadas como las llamadas de conferencia, la grabación de conversaciones o el uso de identificadores personalizados. Estas características han impulsado su adopción masiva en detrimento de las tecnologías tradicionales [1].

En este contexto, es fundamental comprender cómo opera la empresa **BTS**[1], colaboradora en este proyecto, para entender el impacto que podría tener la implementación del firewall basado en eBPF y XDP en su infraestructura. BTS es un proveedor global de servicios de monitoreo y gestión de tráfico VoIP. Opera **plataformas** distribuidas por todo el mundo que incluyen sistemas de intercambio de tráfico telefónico, análisis de calidad, detección de errores y facturación de servicios de voz [2]. En el año 2022, la compañía se posicionó como el **séptimo operador a nivel mundial** en términos de tráfico telefónico internacional, según un informe publicado por *Telegeography* [3].

La arquitectura de estas plataformas se compone de un “Switch de voz” central llamado “Dedalus”, un desarrollo propio de la empresa que funciona como núcleo de sus operaciones. Dicho “Switch de Voz” está compuesto por diversos módulos distribuidos en servidores independientes, como se muestra a modo de esquema en la *Figura 1*, y que gestionan tanto el tráfico telefónico como las estadísticas asociadas. En este tipo de sistemas, es crucial analizar el tráfico en tiempo real para garantizar la calidad del servicio y detectar posibles problemas.

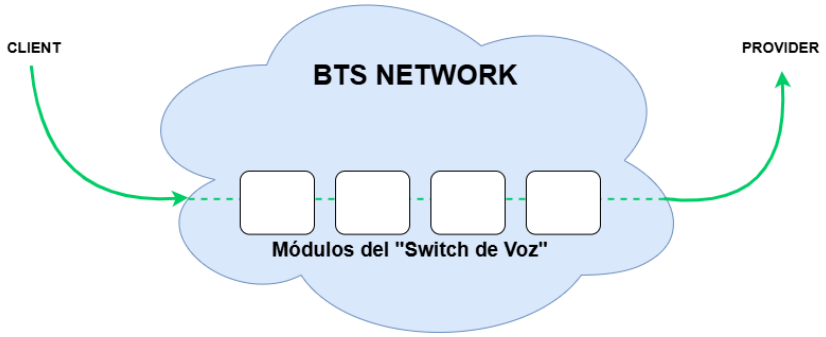


Figura 1. Paso del tráfico telefónico por los módulos de la plataforma.

[1] https://www.bts.io/

Para abordar estos desafíos, las empresas de telecomunicaciones recurren a herramientas de monitoreo y análisis avanzadas. Una parte importante de estas soluciones incluye la virtualización de servidores, donde plataformas como **Proxmox**[2] juegan un papel fundamental. La explicación de esta plataforma se realiza en el *Apartado 2.1*.

En este proyecto, en lugar de las soluciones tradicionales como puede ser ***iptables*** (más detalles en el *Apartado 2.1*, sistema de firewall convencional utilizado en la empresa para analizar tráfico de red en equipos Linux), se explorará el uso de eBPF y XDP como herramientas para implementar un firewall de alto rendimiento. El objetivo es procesar el tráfico SIP de manera más eficiente para mejorar la velocidad, y más eficaz para incrementar la capacidad de análisis. Además, se plantea desarrollar una plataforma web centralizada para gestionar estos firewalls en diversas máquinas de la red de la empresa, lo que permitirá una configuración más sencilla y accesible para los trabajadores.

El siguiente esquema (*Figura 2*) ilustra cómo se integrará el firewall basado en eBPF dentro de la infraestructura de la empresa, destacando los flujos de tráfico entre los diferentes componentes y cómo permitirá analizar tráfico en la entrada a la red.

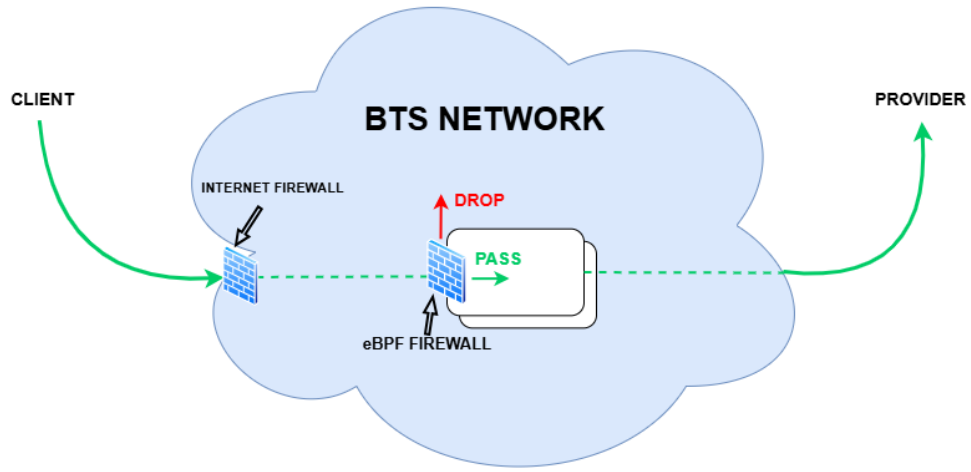


Figura 2. Ubicación del Firewall eBPF en la red.

En definitiva, este proyecto busca evaluar las ventajas de eBPF y XDP frente a las tecnologías tradicionales, así como proponer una solución que permita a BTS optimizar el rendimiento de sus sistemas, facilitando al mismo tiempo la gestión centralizada y mejorando la experiencia de sus clientes en la comunicación VoIP.

[2] https://www.proxmox.com/en/

1.2. Motivación y objetivos

El objetivo principal de este Trabajo Fin de Máster es el estudio y la evaluación de las funcionalidades que proporciona la tecnología **eBPF**, mediante el desarrollo y la implementación de un **firewall Linux** basado en ella. Se persigue el fin de optimizar la gestión y el análisis del tráfico SIP. Esta solución busca superar las limitaciones o funcionalidades limitadas de herramientas tradicionales como *iptables*, ofreciendo una mayor eficiencia y rendimiento en el procesamiento del tráfico en tiempo real.

La hipótesis de este trabajo es que la incorporación de eBPF en los equipos de BTS permitirá optimizar el análisis de paquetes al realizarlo directamente en el espacio del núcleo (en inglés *kernel*), lo que supone una mejora significativa en términos de eficiencia [4]. Esto resulta especialmente valioso en entornos como el de BTS, donde se gestionan grandes volúmenes de tráfico SIP en tiempo real y el rendimiento es un factor crítico para garantizar la calidad del servicio.

Además, como parte del proyecto, se desarrollará una plataforma web centralizada e intuitiva para gestionar los firewalls desplegados en distintos servidores. Esto facilitará la administración y permitirá que el personal acceda a las configuraciones de forma sencilla y eficiente desde un único punto.

Por tanto, los objetivos del proyecto consisten en:

1. **Objetivos generales:**
    a. Comprender y estudiar la tecnología eBPF.
    b. Aplicar la tecnología al desarrollo de un firewall Linux.
    c. Preparar un método de gestión centralizada.
    d. Unificar las partes en un sistema completo y su interacción.
    e. Análisis de comportamiento de eBPF frente a tráfico realista. Comparación con el sistema actual y planteamiento de conclusiones.

2. **Objetivos específicos:**
    a. Estudiar y aplicar distintos lenguajes de programación para el desarrollo del sistema.
    b. Desarrollar una web y lo que ello implica.
    c. Comunicar los distintos programas mediante APIs.
    d. Aplicar políticas de seguridad.
    e. Realizar pruebas de funcionamiento del sistema de gestión y del propio Firewall eBPF.
    f. Realizar pruebas de carga, sometiendo el equipo a altos volúmenes de tráfico.

El esquema del funcionamiento deseado (*Figura 3*) muestra cómo esta implementación permitirá un análisis optimizado, garantizando un impacto positivo en la infraestructura de BTS para los trabajadores.

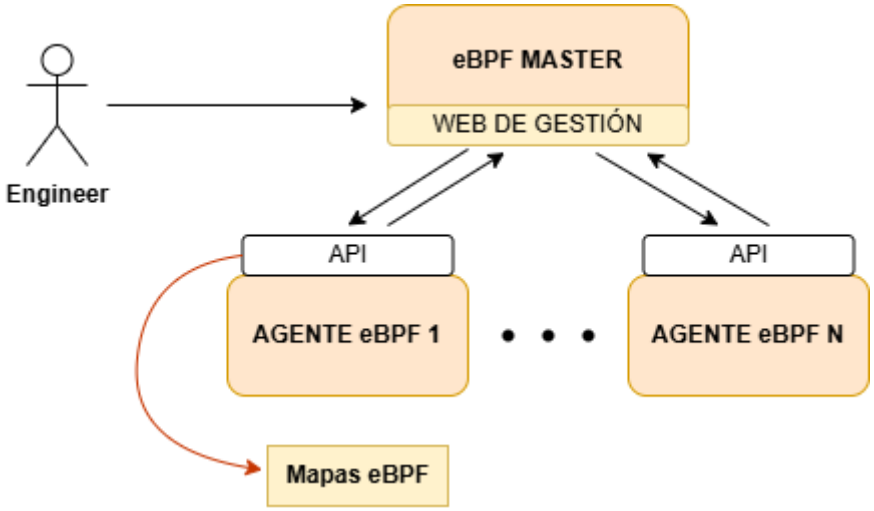


Figura 3. Esquema del funcionamiento del sistema deseado.

1.3. Organización de la memoria

La memoria se presenta de acuerdo con la siguiente disposición:

- Capítulo 1: Se hace una introducción del trabajo, y se destacan las motivaciones y objetivos del mismo.
- Capítulo 2: Se describen los protocolos y herramientas utilizadas, se hace un estudio inicial de la tecnología eBPF y un análisis previo de la situación actual y del funcionamiento de la empresa. Finalmente se plantea una solución para implementar eBPF a un firewall Linux.
- Capítulo 3: Se expone el sistema a desarrollar: Bases del proyecto, entorno de trabajo y tecnologías aplicadas junto a su instalación.
- Capítulo 4: Se realiza la explicación del código de todas las partes que componen el sistema completo.
- Capítulo 5: Se explican y presentan las pruebas realizadas durante el trabajo y se muestran los resultados de las mismas.
- Capítulo 6: Se exponen las conclusiones finales del trabajo y las posibles líneas futuras.

Finalmente, se incluyen varios anexos explicativos al final del documento, que complementan y amplían el contenido de la memoria.

Capítulo 2: Análisis previo

2.1. Protocolos, herramientas y lenguajes utilizados

Este apartado describe los principales protocolos, herramientas y lenguajes de programación que se han utilizado a lo largo del proyecto.

Protocolo SIP

En el contexto del proyecto, basado en el análisis de tráfico telefónico, el protocolo SIP es de particular importancia, ya que se utiliza para establecer y controlar las llamadas telefónicas sobre redes IP. Al analizar el tráfico telefónico basado en SIP, se puede obtener información valiosa sobre las llamadas realizadas, como la duración, origen y destino, servicios utilizados, flujos de datos y otros detalles. Por ello, para comprender los objetivos de este proyecto, es vital conocer cómo funciona este protocolo.

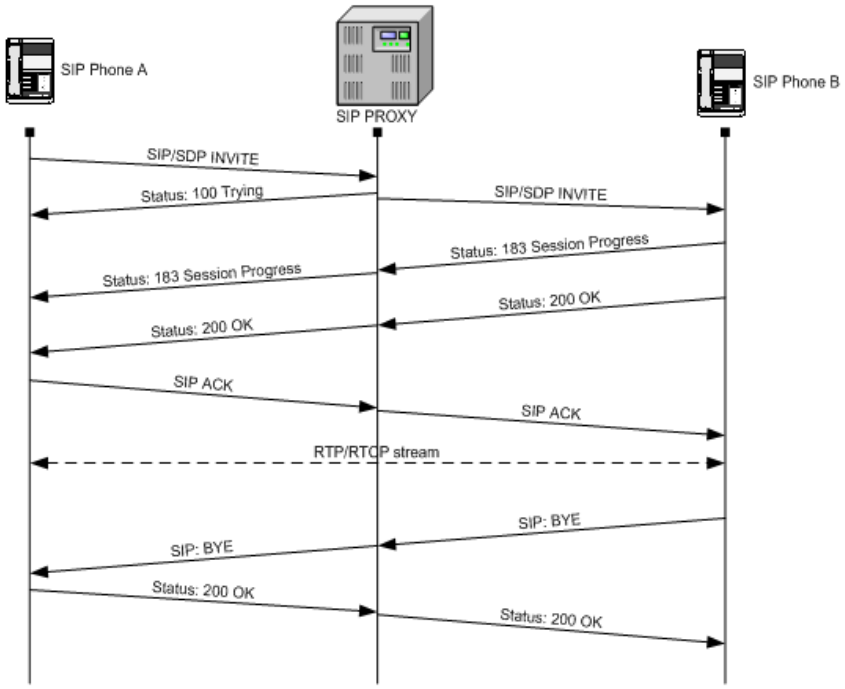


Figura 4. Esquema de funcionamiento del protocolo SIP

El **Protocolo de Inicio de Sesión (SIP)** [5] es un protocolo de señalización utilizado en redes de comunicación de voz y vídeo a través de internet. Fue desarrollado por el IETF (grupo de trabajo de ingeniería de internet), y está definido en el estándar RFC 3261. Información más detallada sobre SIP puede consultarse en el ANEXO A.

Firewall de red - Whitelist

Un firewall de red es una herramienta de **seguridad** que filtra y controla el tráfico que entra y sale de una red, basándose en reglas predefinidas. Su objetivo es proteger los sistemas internos de accesos no autorizados y amenazas externas, como ataques o intrusiones.

Inspecciona los paquetes de datos y decide si permitirlos, bloquearlos o redirigirlos, según criterios como direcciones IP, puertos o protocolos. Además, puede usar **Whitelists (listas blancas)**, listas que especifican direcciones o elementos concretos autorizados para acceder al sistema. Los firewalls, ya sea a nivel de red o aplicación, son esenciales para la seguridad en redes personales y corporativas.

Firewall con iptables

iptables es una herramienta en Linux utilizada para configurar políticas de filtrado y manejo de tráfico de red en el *kernel* [6]. Opera como parte del subsistema *netfilter*, que se encarga de inspeccionar y manipular paquetes de red en diferentes puntos de su recorrido (entrada, salida o reenvío).

Las reglas en *iptables* definen cómo debe manejarse un paquete según criterios como dirección IP, puerto, protocolo, y más. Estas reglas se agrupan en cadenas (chains) como INPUT, OUTPUT, y FORWARD, que forman parte de tablas específicas.

eBPF

eBPF (por sus siglas en inglés: *Extended Berkeley Packet Filter*)[3] es una tecnología moderna que permite **ejecutar programas personalizados de forma segura dentro del núcleo (*kernel*)** de un sistema operativo, sin necesidad de modificarlo directamente. Originalmente diseñada para el filtrado de paquetes de red, eBPF ha evolucionado significativamente, encontrando aplicaciones en monitoreo, seguridad, optimización de redes y otros ámbitos del sistema.

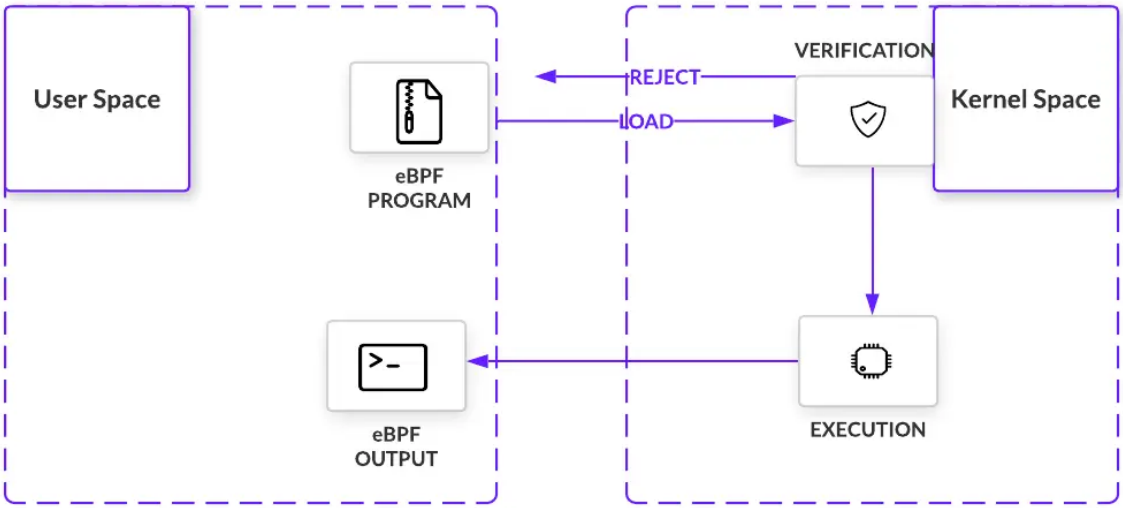


Figura 5. Esquema simplificado de funcionamiento de eBPF

[3] https://ebpf.io/

La principal ventaja de eBPF es que combina el alto rendimiento de las operaciones en el núcleo con la **flexibilidad** de ejecutar programas definidos por el usuario. Esto se logra al permitir que pequeños programas en eBPF sean cargados en puntos específicos del *kernel*, donde se ejecutan de manera controlada y segura. Estos programas se validan antes de ejecutarse para garantizar que no comprometan la estabilidad del sistema.

Gracias a su diseño, eBPF se utiliza ampliamente en la observación y administración de sistemas tecnológicos modernos, permitiendo implementar soluciones avanzadas como firewalls, herramientas de monitoreo en tiempo real y análisis de rendimiento detallado. Su capacidad para interactuar con datos a nivel del sistema lo convierte en una herramienta poderosa para desarrolladores e ingenieros de redes.

Se detallan más aspectos técnicos y específicos de esta tecnología en el *Apartado 2.2*.

XDP

XDP (*eXpress Data Path*) es una tecnología diseñada para **procesar paquetes** de red de manera extremadamente eficiente, **directamente en la interfaz de red**. Es una extensión de eBPF que aprovecha su capacidad para ejecutar programas en el *kernel* de un sistema operativo, pero con un enfoque específico en la manipulación y gestión de paquetes de red a alta velocidad. [7]

Lo que hace único a XDP es que actúa en la etapa más temprana posible del procesamiento de paquetes, **antes de que estos entren al stack de red del sistema operativo**. Esto se logra **cargando programas eBPF directamente en la interfaz de red**, lo que permite decisiones rápidas sobre los paquetes, como permitir su paso (pass), bloquearlos (drop) o incluso reenviarlos (retransmit). Este nivel de control permite **reducir significativamente la latencia y el consumo de recursos del sistema**, ya que los paquetes que no son necesarios pueden ser descartados antes de ser procesados más a fondo.

Las principales funciones que XDP habilita incluyen las siguientes, y pueden observarse en el esquema de la *Figura 6*:

- **PASS**: Permitir que un paquete continúe su curso hacia el stack de red.
- **DROP**: Bloquear paquetes específicos, evitando que lleguen al sistema.
- **REDIRECT**: Redirigir paquetes hacia otra interfaz o destino específico, optimizando rutas de red.

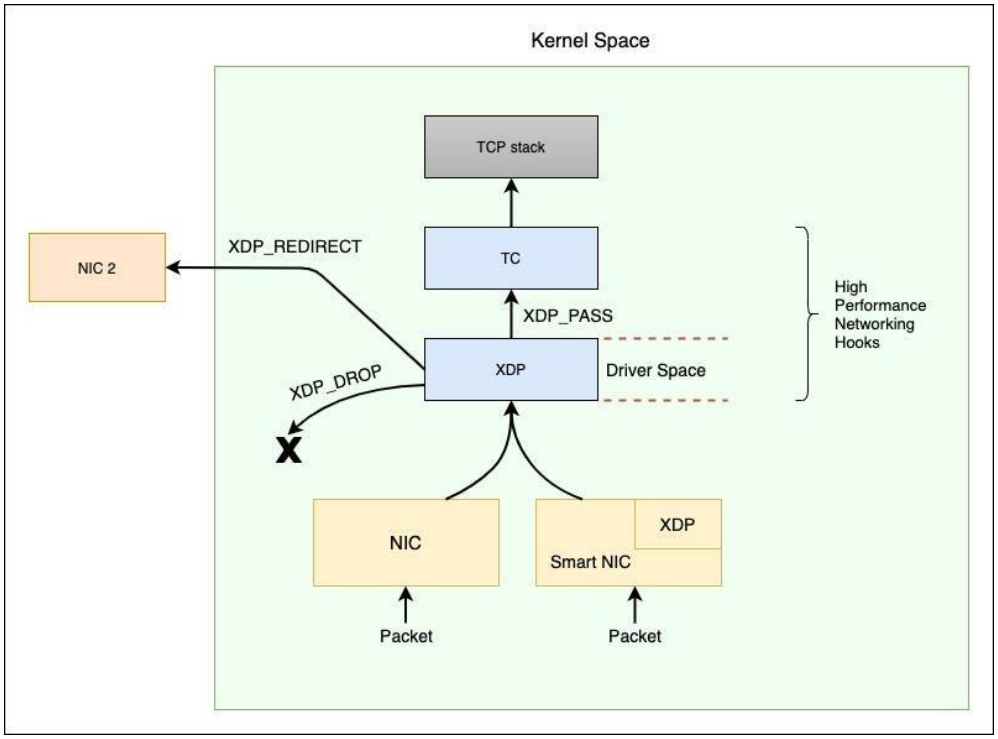


Figura 6. Proceso de decisión con XDP

Gracias a su integración con eBPF, los programas XDP son altamente personalizables y seguros, ya que el sistema valida su ejecución antes de cargarlos. Esto convierte a XDP en una herramienta ideal para casos como firewalls de alta velocidad, mitigación de ataques DDoS y enrutamiento eficiente en redes.

Proxmox Virtual Environment

Proxmox VE (*Virtual Environment*) es una solución empresarial de código abierto para el despliegue de un entorno de virtualización. Su objetivo es ayudar a optimizar el uso de los recursos ya existentes, minimizar el costo por hardware y el tiempo empleado. Utiliza habitualmente Debian GNU/Linux como sistema operativo mediante un *Kernel* de Linux personalizado. Además, permite su instalación sobre un Debian ya existente.

Linux - Kernel

Linux es un **sistema operativo de código abierto** que sirve como base para muchas distribuciones (como Debian, Ubuntu o CentOS). Es ampliamente utilizado en servidores, dispositivos embebidos y computadoras personales debido a su flexibilidad, estabilidad y seguridad.

El núcleo o ***kernel* de Linux**[4] es el componente central del sistema operativo, responsable de gestionar los recursos del hardware, como la memoria, el procesador y los dispositivos periféricos, así como de proporcionar una **interfaz entre el hardware y el software**.

[4] https://www.kernel.org/

Debian

Debian[5] es un sistema operativo basado en Linux, conocido por su estabilidad, seguridad y versatilidad. Es una distribución gratuita y de **código abierto** que incluye un núcleo Linux y un amplio conjunto de software, como herramientas de desarrollo, aplicaciones de oficina y servidores.

Python

Python[6] es un lenguaje de programación de alto nivel, versátil y fácil de aprender, ampliamente utilizado en desarrollo web, análisis de datos, inteligencia artificial y más. Es conocido por su sintaxis clara, que facilita escribir y leer código, y por su extensa biblioteca estándar, que permite realizar diversas tareas sin necesidad de herramientas externas.

API - Flask

Las APIs (Application Programming Interfaces) son conjuntos de reglas y protocolos que permiten a diferentes aplicaciones comunicarse entre sí. Actúan como intermediarios que facilitan el intercambio de datos o funcionalidades entre sistemas, ya sea dentro de un mismo programa o entre aplicaciones distribuidas. Por ejemplo, una API puede permitir que una aplicación móvil consulte datos de un servidor remoto.

En este contexto, **Flask**[7] es una plataforma de desarrollo web en Python que facilita la creación de APIs y aplicaciones web. Es ligero, modular y muy flexible, lo que lo hace ideal tanto para proyectos pequeños como para aplicaciones más complejas. Flask incluye herramientas básicas para gestionar rutas, manejar peticiones HTTP (GET, POST, etc.) y renderizar páginas dinámicas, pero también permite integrar extensiones para añadir funcionalidad avanzada, como autenticación o acceso a bases de datos.

C/BCC

El lenguaje C es un lenguaje de programación de bajo nivel diseñado para desarrollar sistemas operativos y software de alto rendimiento. Es ampliamente utilizado por su eficiencia y su capacidad para interactuar directamente con el hardware. En el caso de los programas eBPF, el **código C se utiliza para escribir los fragmentos de código que serán ejecutados en el *kernel*** del sistema, ya que su naturaleza eficiente y directa lo hace ideal para esta tarea. Estos programas se compilan en ***bytecode***, un formato que puede ser verificado y ejecutado de manera segura dentro del *kernel*.

Para facilitar el desarrollo de programas eBPF, se emplea **BCC (BPF Compiler Collection),** un conjunto de herramientas que simplifica la escritura, compilación y carga de código eBPF en el *kernel*. BCC proporciona bibliotecas y utilidades que permiten desarrollar programas eBPF en lenguajes como Python o C, permitiendo una interacción más accesible entre los programas de usuario y el *kernel*.

[5] https://www.debian.org/
[6] https://www.python.org/
[7] https://flask.palletsprojects.com/en/stable/

HTML

HTML (HyperText Markup Language) es el lenguaje estándar utilizado para **estructurar y presentar contenido en la web**. Permite definir elementos como encabezados, párrafos, enlaces, imágenes y formularios, formando la base de cualquier página web. Es esencial para el diseño y desarrollo de interfaces visuales, y funciona en conjunto con CSS y JavaScript para crear sitios dinámicos y atractivos.

JSON

JSON (JavaScript Object Notation)[8] es un formato de texto ligero y estructurado para **almacenar e intercambiar datos**. Es ampliamente utilizado debido a su simplicidad y facilidad de lectura tanto por humanos como por máquinas. JSON **organiza la información en pares clave-valor** y estructuras como objetos y listas, siendo compatible con la mayoría de lenguajes de programación.

Ejemplo de dato en formato JSON:

```
{
  "nombre": "Juan",
  "edad": 30,
  "ciudad": "Madrid",
  "intereses": ["programación", "música", "viajes"]
}
```

Este formato es común en APIs y **transferencias de datos** en aplicaciones web.

PostgreSQL

PostgreSQL[9] es un sistema de **gestión de bases de datos relacional** de código abierto, conocido por su robustez, flexibilidad y características avanzadas. Es compatible con estándares SQL y permite almacenar y consultar datos estructurados de manera eficiente. PostgreSQL soporta tipos de datos personalizados, índices avanzados, y extensiones para añadir funcionalidad adicional, como el manejo de datos geoespaciales.

Es ampliamente utilizado en aplicaciones web y empresariales debido a su capacidad para **manejar grandes volúmenes de datos y garantizar integridad y consistencia**. Por ejemplo, se puede usar para gestionar una base de datos de usuarios.

Varios comandos relacionados y utilizados en el proyecto pueden consultarse en el ANEXO C.

[8] https://www.json.org/json-es.html
[9] https://www.postgresql.org/

SIPp

SIPp[10] es una herramienta de código abierto de **prueba de rendimiento mediante generación de tráfico para el protocolo SIP**. Incluye algunos escenarios básicos de agentes de usuario (UAC y UAS), y establece y libera múltiples llamadas con los métodos INVITE y BYE. Además, puede leer archivos *.xml* de escenarios personalizados que pueden describir flujos más complejos de llamadas.

También cuenta con la visualización dinámica de estadísticas sobre la ejecución de pruebas (tasa de llamadas, retrasos, estadísticas de mensajes…), volcados periódicos de estadísticas y un ajuste de la tasa de llamadas dinámico.Se puede consultar el modo de uso de la herramienta junto con algún ejemplo de su funcionamiento en el ANEXO I.

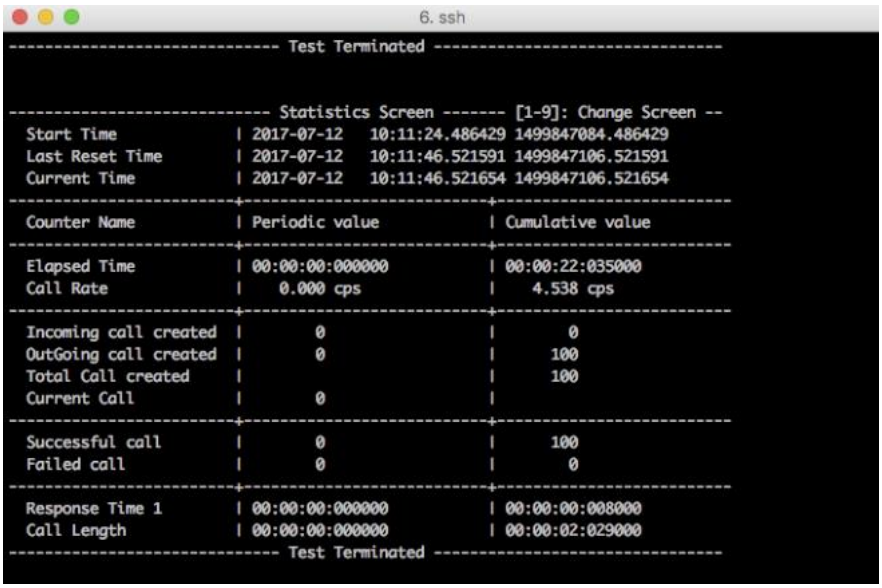


Figura 7. Ejemplo de visualización de la herramienta.

[10] https://sipp.sourceforge.net/

2.2. Estudio y análisis de la tecnología eBPF

El interés de BTS por explorar la tecnología **eBPF** radica en su potencial para revolucionar la forma en que se analizan y gestionan los sistemas de redes informáticas. eBPF permite la ejecución de código **directamente en el *kernel* de Linux**, lo que abre la puerta a un mayor rendimiento y eficiencia en cualquier entorno tecnológico en el que se aplique. Como se ha comentado anteriormente, para una empresa que debe gestionar altos volúmenes de tráfico SIP (entre otros) en tiempo real, eBPF se perfila como una herramienta innovadora y muy interesante para optimizar procesos críticos como el filtrado y análisis de este tráfico en tiempo real.

2.2.1. Historia y fundamentos de eBPF

eBPF (por sus siglas en inglés: *Extended Berkeley Packet Filter*) [8] tiene sus raíces en BPF, una tecnología introducida en 1992 como una herramienta para capturar y filtrar paquetes en sistemas Unix. Diseñado **inicialmente** para el **análisis eficiente de paquetes en redes**, BPF era limitado en su funcionalidad, centrándose exclusivamente en operaciones relacionadas con paquetes de red. En 2014, con la versión 3.18 del *kernel* de Linux, **BPF evolucionó a eBPF**, ampliando enormemente su alcance y utilidad. Ahora, eBPF no solo se utiliza para el análisis de paquetes, sino también para el **monitoreo avanzado del sistema, control del tráfico en tiempo real, y ejecución de programas personalizados dentro del *kernel***.

La característica más notable de eBPF es su capacidad para permitir que programas seguros y optimizados se ejecuten **dentro del *kernel* de Linux** sin necesidad de modificar el propio código del *kernel*, algo históricamente complejo y propenso a errores. eBPF logra esto gracias a un verificador que analiza y asegura la seguridad de cada programa antes de su ejecución, protegiendo la estabilidad del sistema.

Principales Ventajas de eBPF

- Eficiencia: Al operar directamente dentro del *kernel*, eBPF elimina las transiciones entre el espacio de usuario y el espacio del *kernel*, minimizando la latencia y reduciendo significativamente la sobrecarga del sistema.
- Flexibilidad: Permite personalizar y ejecutar funciones avanzadas para tareas como monitoreo de aplicaciones, gestión de redes, y políticas de seguridad sin interrumpir el funcionamiento normal del sistema.
- Seguridad: Los programas eBPF deben pasar por un verificador estricto, asegurando que no puedan causar inestabilidad o comprometer la integridad del *kernel*.
- Escalabilidad: Diseñado para manejar grandes volúmenes de tráfico y entornos de alta concurrencia, eBPF es ideal para aplicaciones en tiempo real y sistemas distribuidos.

Los programas eBPF se escriben generalmente en **C**, un lenguaje que permite un control detallado del hardware y el *kernel*. Estos programas se compilan utilizando herramientas como *LLVM/Clang*, que traducen el código C a *bytecode*, un formato que puede ser entendido y ejecutado por el verificador de eBPF en el *kernel*. Además, para facilitar el desarrollo, herramientas como **BCC** permiten a los desarrolladores escribir y ejecutar programas eBPF

utilizando lenguajes como Python, combinando la potencia de eBPF con la simplicidad de lenguajes más accesibles.

Una de las características más poderosas de eBPF son los **"mapas eBPF"**, estructuras de datos que permiten la **comunicación entre los programas eBPF y el espacio de usuario**. Los mapas son versátiles y eficientes, utilizados para almacenar información clave, como estadísticas, listas de reglas de firewall, o datos temporales necesarios durante la ejecución de un programa eBPF.

2.2.2. Uso actual en la industria[11]

La tecnología eBPF ha revolucionado la industria tecnológica al permitir la ejecución de programas en espacio aislado dentro del *kernel* de Linux, ampliando sus capacidades. Entre las aplicaciones principales que está tomando en la industria tecnológica mundial, podemos destacar lo siguiente:

- **Observabilidad y Monitoreo**: eBPF permite a los desarrolladores instrumentar el *kernel* y las aplicaciones de espacio de usuario para recopilar datos y métricas de rendimiento detallados sin afectar significativamente el rendimiento del sistema.
- **Seguridad**: Proyectos como *Falco* utilizan eBPF para la monitorización de seguridad, detectando comportamientos anómalos en tiempo real.
- **Redes y Balanceo de Carga**: Empresas como *Facebook* han desarrollado proyectos como *Katran*, que emplea eBPF para crear un balanceo de carga L4 más eficiente.

En resumen, eBPF se ha consolidado como una herramienta esencial en la industria tecnológica, facilitando soluciones avanzadas en observabilidad, seguridad y redes, y promoviendo un ecosistema de innovación y eficiencia en sistemas modernos. [9]

Información ampliada sobre proyectos, patentes o empresas que utilizan eBPF puede consultarse en el ANEXO B.

[11] https://ebpf.io/es/case-studies/

2.3. Sistema actual y planteamiento del problema

El tráfico que cursa BTS en su mayoría es tráfico SIP [10] (cuyo funcionamiento básico se explica en el *Apartado 2.1*), que se redirecciona a la ruta más efectiva para que las llamadas se ejecuten con la máxima eficiencia y calidad de servicio. Como se puede observar en la *Figura 8*, en esencia, las llamadas VoIP vienen desde un cliente (origen del tráfico) por una conexión o "Trunk" SIP de entrada hasta el equipo de interconexión dentro de la red de BTS, encargado de proporcionar el control de la llamada, procesamiento, y otros servicios sobre la red de conmutación de paquetes IP (**Switch de voz**).

Este Switch de Voz es el encargado de decidir por qué proveedor conecta la llamada que procesa, a través de un "Trunk" SIP de salida. La llamada involucra varios diálogos SIP, que se gestionan con los distintos servidores y equipos de la plataforma.

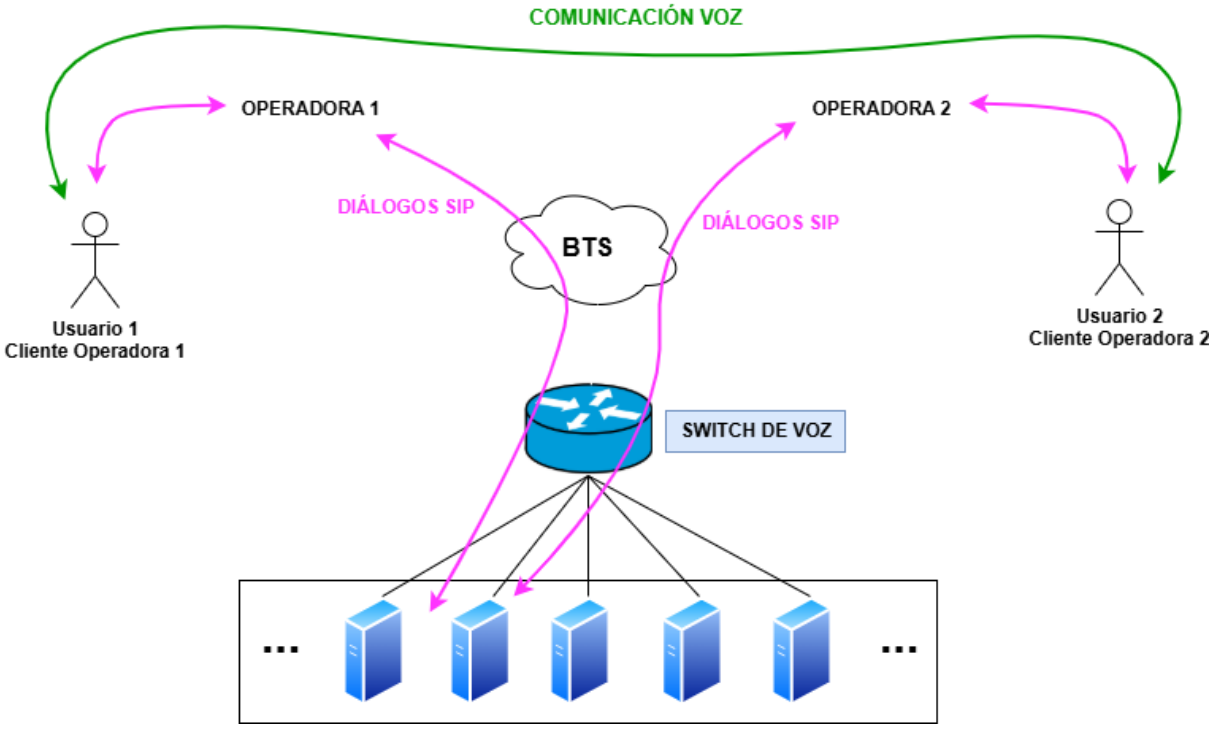


Figura 8. Intercambio de mensajes en la comunicación SIP

BTS, que utiliza la tecnología de voz sobre IP, permite a los usuarios realizar llamadas a través de una conexión de internet de alta velocidad en lugar de una línea telefónica tradicional. Los clientes de BTS (operadoras de cualquier país) se comunican con la compañía, para que les proporcione el soporte y conecte las llamadas con operadoras distintas o incluso de otros países. Es posible que BTS no sea proveedor del destino de una comunicación, en cuyo caso pasará a ser cliente de otro proveedor, que sí pueda establecer comunicación con la operadora final, y poderse realizar la llamada o llamadas. Por tanto, funciona a la vez como proveedor de servicios a compañías que contactan con ellos (denominadas **clientes**) y como cliente de otros proveedores telefónicos que consigan que se dé la comunicación con el destinatario (denominados **proveedores**).

Actualmente, el flujo de paquetes de las comunicaciones que pasan por BTS está definido y utiliza **diversos módulos de la plataforma** o "Switch de Voz", que realizan funciones distintas sobre el tráfico. Entre estos módulos (que son desarrollos propios de la empresa) se pueden destacar los siguientes:

- **SIP Dedge**: Receptor inicial del tráfico. Balanceo de carga hacia los siguientes equipos.
- **SIP Proxy IN – Talos**: Proxy de entrada. Autentica al cliente y procesa el tráfico SIP, por ejemplo, quitando o añadiendo cabeceras.
- **Minos / Niso**: Coordina la salida del tráfico SIP hacia los Theseus.
- **SIP Proxy OUT – Theseus:** Proxy de salida. Enruta el tráfico hacia el exterior.

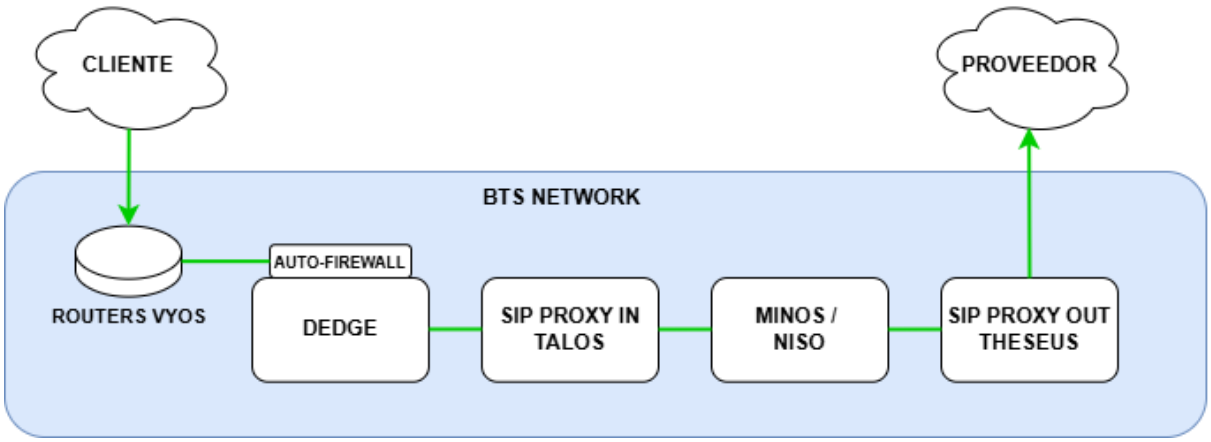


Figura 9. Curso del tráfico a través de los módulos principales.

Todo el procesamiento del tráfico que se lleva a cabo dentro de la red de la empresa tiene como etapa previa un elemento fundamental en cualquier infraestructura de red: el firewall en la salida hacia Internet. Este firewall es esencial, ya que define qué tráfico está permitido o bloqueado, protegiendo los equipos internos de accesos no deseados. Sin él, cualquier tráfico de cualquier origen podría ingresar en la red interna, con consecuencias potencialmente graves para la seguridad. En el caso de BTS, este firewall se implementa en los routers VyOS, los dispositivos más potentes de la infraestructura, responsables de analizar y proteger el tráfico en una primera línea de defensa.

Aunque el sistema de firewall basado en los routers VyOS es eficaz y seguro, su eficiencia se ve limitada en escenarios donde es necesario actualizar las políticas de manera frecuente. Este es el caso del tráfico SIP en BTS, donde las comunicaciones con proveedores y clientes requieren ajustes constantes. Cada vez que se acuerda una nueva interconexión, es necesario configurar las direcciones IP específicas indicadas por el otro lado para permitir la comunicación. Esta situación implica realizar modificaciones continuas en los firewalls de VyOS, lo que puede no solo aumentar el riesgo de errores en la configuración, sino también reducir la eficiencia operativa.

Para abordar este desafío en la organización y gestión del tráfico, surge el proyecto interno "**Auto-Firewall**". Este enfoque simplifica la administración al permitir todo el tráfico dirigido al puerto 5060 (destinado al tráfico SIP) en los routers VyOS, reduciendo significativamente la necesidad de intervenciones frecuentes en estos dispositivos. En lugar de bloquear el tráfico no deseado directamente en los routers, **el filtrado se realiza en los primeros equipos del "Switch de Voz": los "Dedge"**. Esto optimiza tanto la seguridad como la eficiencia en la gestión del tráfico SIP.

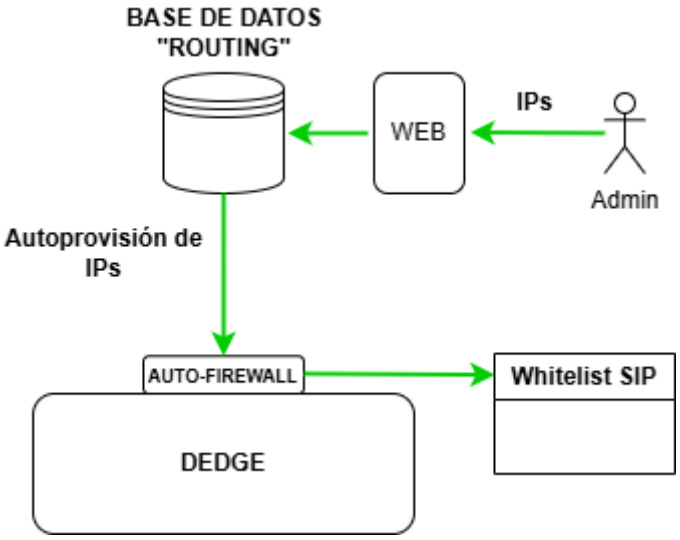


Figura 10. Funcionamiento del Auto-Firewall y Routing.

Actualmente, los "Dedge", que en la práctica actúan también como firewalls, se auto provisionan a partir de una base de datos interna de la empresa llamada "**Routing**" – como se muestra en la *Figura 10.* En esta base de datos se almacenan las direcciones IP de clientes y proveedores que deben ser autorizadas para acceder a la red interna, tal y como se ilustra en la *Figura 11*. La gestión de estas IP se realiza a través de una interfaz web diseñada para facilitar su administración, haciendo que el proceso sea más sencillo e intuitivo para los usuarios. El mecanismo consiste en recopilar todas estas direcciones IP desde la base de datos y utilizarlas para generar automáticamente una lista de permisión (*whitelist*) en el firewall de la máquina Linux, asegurando que solo el tráfico autorizado tenga acceso a la red interna y siga con el proceso habitual.

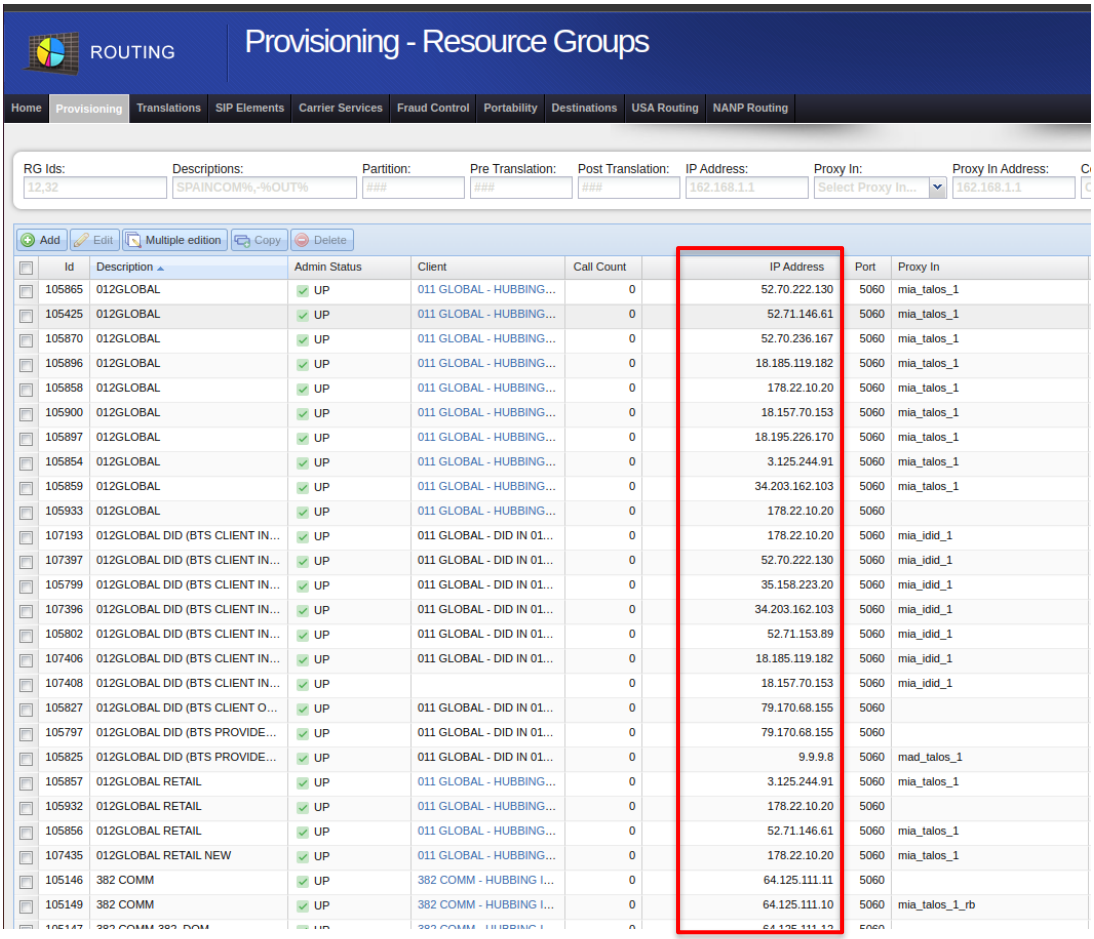


Figura 11. Web de la base de datos "Routing".

Como se ha mencionado, el tráfico gestionado por BTS atraviesa diversos equipos y sistemas que realizan múltiples funciones sobre las comunicaciones. Este procesamiento, combinado con la elevada cantidad de tráfico bruto manejado, puede ocasionar problemas de latencia debido al alto nivel de tratamiento que requieren los paquetes. Optimizar la eficiencia en estos procesos es una prioridad constante para BTS, y en este contexto, la tecnología eBPF emerge como una solución prometedora para abordar estas dificultades.

Tras evaluar las posibles aplicaciones de eBPF en la infraestructura actual, se ha decidido aplicar esta tecnología al actual "Auto-Firewall", operando en los equipos "Dedge" Linux que actúan como firewall interno. Si los resultados de esta implementación resultasen favorables, el nuevo firewall con eBPF podría reemplazar al sistema actual, ofreciendo mejoras significativas en términos de rendimiento y gestión del tráfico.

2.4. Propuesta de solución: Firewall Linux con eBPF

Tras el proceso de estudio de la tecnología eBPF, surgieron varias vías de aplicación, a partir de las cuales se decidió montar un **Firewall Linux con eBPF** ya que su aplicación en la infraestructura sería directa y podría mostrar resultados respecto al uso de recursos y eficiencia de los sistemas. Además, abre un abanico de posibilidades extra que buscan acercar el manejo de estas herramientas a todos los trabajadores.

El firewall Linux desarrollado con eBPF tiene como objetivo principal mejorar la eficiencia y optimizar el uso de recursos en los equipos. Para ello, aprovecha las ventajas de esta tecnología en el sistema actual que actúa como firewall para el tráfico SIP (denominado "Auto-Firewall", resumido en la *Figura 10*).

A continuación, se comentan las principales características con las que contará este nuevo Firewall eBPF, y que conformarán el sistema a desarrollar.

- **Aplicación de XDP**

XDP, como se ha comentado en el *Apartado 2.1*, es una tecnología de Linux diseñada para procesar paquetes de red directamente en la **NIC** (tarjeta de red de un equipo) o en la **capa más baja del stack de red del *kernel***. eBPF permite la integración en sus programas personalizables con XDP (*Figura 12)*. Esto es altamente beneficioso ya que la asociación de un programa eBPF directamente a un interfaz de red permite realizar el proceso habitual de un firewall (permitir o tirar tráfico) en primera instancia y sin necesidad de pasar los paquetes a las diferentes capas del sistema.

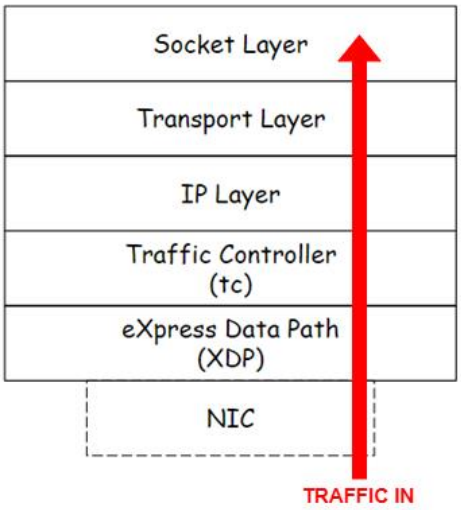


Figura 12. Punto de acción de XDP en un equipo.

- **Mismas funcionalidades que *iptables***

El servicio *iptables* (comentado resumidamente en el *Apartado 2.1)* es una herramienta en Linux utilizada para configurar reglas de firewall mediante el filtrado de paquetes. Funciona como una interfaz para controlar el framework de *Netfilter* (integrado en Linux), permitiendo definir reglas para aceptar, rechazar o redirigir tráfico de red según criterios específicos como direcciones IP, puertos o protocolos.

Iptables es el sistema que BTS utiliza actualmente como firewall en sus máquinas Linux. Aunque poderoso, puede ser menos eficiente en entornos de alto tráfico debido a su procesamiento secuencial de reglas. Ya que *iptables* está configurado con reglas que funcionan actualmente en los equipos, el Firewall eBPF será capaz de replicar este funcionamiento actual, de forma que pudiera ser aplicado sin afección en los sistemas. Un ejemplo de firewall con *iptables* puede ser consultado en la *Figura 13*, donde se aprovecha para remarcar las líneas relativas a las *whitelist*.

Figura 13. Ejemplo de Firewall iptables en producción.

- **Centralización de la gestión**

Otra cualidad del proyecto desarrollado es conseguir la centralización de la gestión de los firewall en las distintas máquinas Linux con las que se trabaja. Instalando el Firewall eBPF en varias máquinas, podrán ser configuradas desde un punto central, denominado "eBPF Master". Desde él, se podrá escoger el agente concreto que configurar para añadir, listar o eliminar reglas de firewall de una manera eficiente manteniendo las funcionalidades actuales.

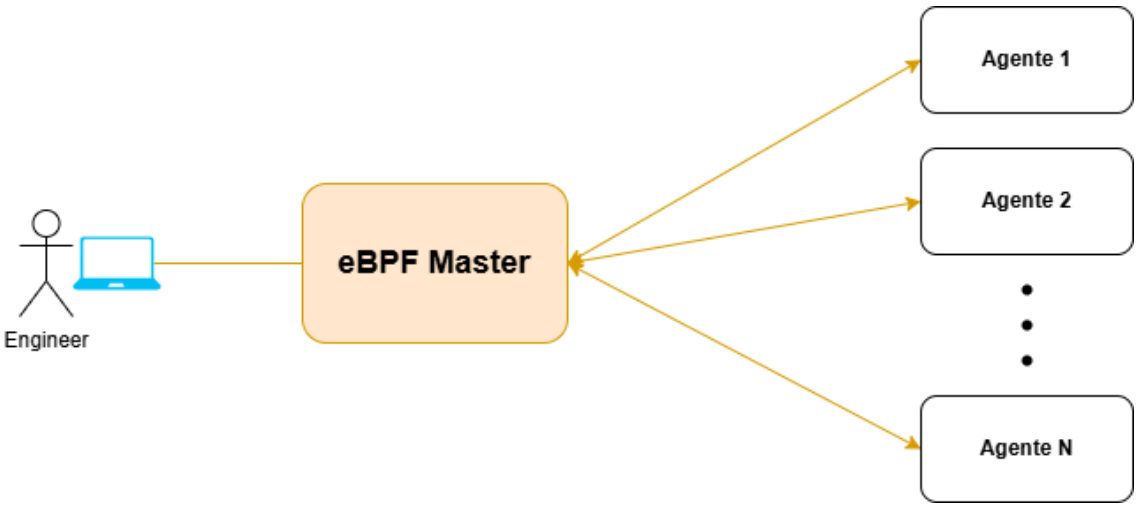


Figura 14. Esquema de la gestión centralizada.

- **Gestión Web de los Agentes eBPF**

Como complemento a la gestión centralizada de los diferentes agentes en los que funciona eBPF, se prepara una web simple que permita realizar las tareas básicas del firewall de una forma sencilla. Desde esta web se podrá añadir nuevas reglas al firewall, eliminar reglas existentes y listar las reglas que se apliquen en el agente.

- **Definición de reglas por defecto**

El firewall con eBPF, de igual forma que el sistema *iptables* que funciona actualmente, cuenta con la funcionalidad de cargar reglas por defecto al levantar el programa desde cero. Es decir, una vez se carga el programa eBPF en el agente final, se contará con un conjunto de reglas básicas que permiten la configuración sin problema y sin perder acceso al sistema.

Esta funcionalidad es personalizable (se pueden elegir de forma sencilla las reglas iniciales) e indispensable para situaciones como el reinicio del sistema o del propio firewall.

- **Aplicación instantánea de políticas**

En los agentes en los que se aplique el Firewall eBPF, se podrá tener un manejo en tiempo real de las políticas de tráfico. Si se decide añadir o eliminar una regla de firewall desde la web centralizada, el cambio tendrá efecto en el mismo momento, sin necesidad de reiniciar el programa eBPF o realizar cualquier acción en el agente final.

- ***Whitelist* para tráfico SIP**

Como se ha explicado anteriormente, el sistema de firewall actual en las máquinas Linux utiliza una *whitelist* para permitir el tráfico proveniente de las IPs que la conforman y descartar el resto. El Firewall eBPF también cuenta con esta funcionalidad indispensable para la empresa, y de igual forma que el actual *Auto-Firewall*, es capaz de leer la *whitelist* desde una base de datos externa.

De esta forma, el programa eBPF es capaz de actualizar sus políticas en tiempo real auto provisionándose de esta *whitelist* que se ubica en una base de datos, y que se gestiona externamente.

- **Comunicación segura entre el Master y los agentes finales**

Una parte esencial del proyecto y más en concreto de la comunicación entre los diferentes protagonistas del sistema es la **seguridad**.

Por un lado, la comunicación entre el trabajador (en este caso, ingeniero de BTS) y la web de gestión de agentes debe estar protegida. Esto se implementa mediante la aplicación de certificados TLS en la web, y un sistema de Login que protege el sistema de accesos no autorizados. Por otro lado, la comunicación entre la web de gestión y el propio programa eBPF que se aplica en el firewall de los agentes, también debe ser protegida. Para ello, se aplican certificados TLS a la comunicación entre las APIs que interactúan. Todo ello queda descrito en la *Figura 15*.

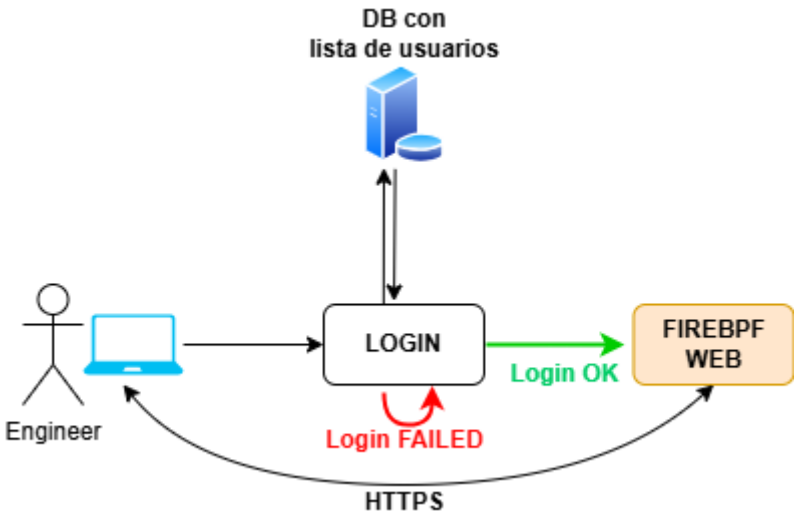


Figura 15. Seguridad aplicada al sistema

Capítulo 3: Exposición del proyecto a desarrollar

3.1. Descripción y bases del proyecto

Como se ha mencionado en el apartado anterior, el proyecto se centra en aplicar las capacidades de la tecnología eBPF al sistema de gestión de tráfico SIP que actualmente utiliza la empresa. El objetivo principal es replicar el funcionamiento del firewall en uso en los equipos de comunicación para evaluar las posibles mejoras que esta tecnología pueda aportar.

El sistema actualmente en producción, denominado "Auto-Firewall", se basa en la aplicación de una *whitelist* al tráfico SIP que ingresa a la infraestructura de la empresa. Esto permite que el primer equipo en la ruta del tráfico, los denominados "Dedge", determine si dicho tráfico debe ser aceptado o rechazado antes de que sea procesado por los sistemas internos.

En este contexto, eBPF resulta especialmente interesante, ya que los equipos deben manejar grandes volúmenes de tráfico, lo que puede generar una sobrecarga en sus recursos. Si se logra mejorar la eficiencia reemplazando el actual uso de *iptables* con eBPF, se podrían optimizar recursos críticos como la CPU y la memoria, elementos fundamentales en un entorno de comunicaciones de alto rendimiento como el de BTS.

Además de las mejoras en eficiencia, eBPF ofrece un alto grado de flexibilidad, permitiendo personalizar completamente los programas que se ejecutan. Esta característica habilita el desarrollo de un sistema de gestión centralizado basado en una interfaz web, lo que facilita a los empleados la administración del Firewall eBPF de manera intuitiva y efectiva, optimizando tanto el tiempo como el esfuerzo requerido para las tareas de gestión.

Otro punto importante por destacar del desarrollo del firewall es el uso de **APIs**. Las APIs son interfaces que permiten la comunicación y el intercambio de información entre los distintos componentes del sistema, en este caso, entre la aplicación web de gestión y el programa eBPF. A través de "endpoints" definidos, las APIs permiten realizar operaciones como la actualización de políticas de filtrado y la configuración dinámica del firewall sin necesidad de acceso manual a los equipos. Este enfoque proporciona una interacción centralizada, eficiente y segura, además de facilitar la escalabilidad del sistema al permitir la integración con futuros módulos o herramientas de gestión.

Con todo ello, el objetivo final es, por un lado, evaluar las capacidades y funcionalidades que ofrece la tecnología eBPF y, por otro, desarrollar un entorno funcional que pueda integrarse en producción, mejorando la accesibilidad y facilitando su uso por parte de los trabajadores.

3.2. Entorno de trabajo

Este proyecto es desarrollado en el entorno de una empresa de telecomunicaciones, BTS. La compañía trabaja en su mayoría con servidores propios, distribuidos en distintos puntos como Madrid, Miami o Alemania. Estos servidores de trabajo pueden ser físicos o virtuales, siendo estos últimos los que predominan cada vez más. La virtualización de servidores en un entorno de comunicaciones como el de BTS ofrece numerosas ventajas, especialmente en términos de flexibilidad, eficiencia y optimización de recursos. Al permitir la ejecución de múltiples máquinas virtuales en un solo servidor físico, se maximiza el uso del hardware disponible, reduciendo costos operativos y de mantenimiento. Además, la virtualización facilita la creación de entornos de pruebas, la implementación rápida de nuevas soluciones y la escalabilidad del sistema, adaptándose de manera eficiente a las demandas variables del tráfico de comunicaciones.

En este contexto, se opta por desarrollar el proyecto en **máquinas virtualizadas**, ya que los equipos finales previstos para el despliegue del Firewall eBPF en producción (los “Dedge”) también operan en entornos virtualizados. La conexión a estas máquinas virtualizadas, al igual que al resto de la infraestructura, se realiza a través de una VPN corporativa. Este sistema permite a los trabajadores de BTS obtener una IP privada dentro de la red de la empresa, proporcionando acceso seguro a las máquinas internas a través del servidor principal de VPN, ubicado en Madrid. El esquema de la *Figura 16* ilustra este funcionamiento.

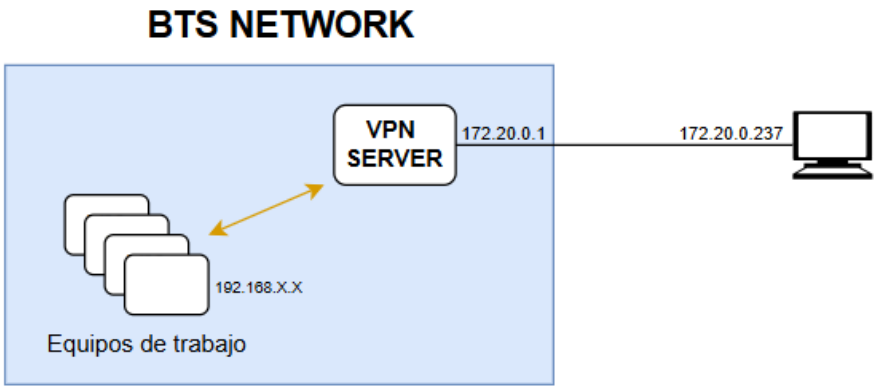


Figura 16. Entorno de trabajo y conexión VPN.

La empresa cuenta además con un entorno de "desarrollo" alojado en servidores proporcionados por **Hetzner**, un proveedor de servicios de hosting de alto rendimiento. Estos equipos, ubicados en Alemania, no se emplean para tareas de producción, pero resultan ideales para preparar entornos de pruebas o desarrollo. Gracias a ello, se convierten en una plataforma adecuada para realizar las pruebas de este proyecto, garantizando que las actividades no afecten al funcionamiento operativo de la empresa y permitiendo prácticas aisladas. El esquema básico de conexión con los equipos de Hetzner se muestra en la *Figura 17* a continuación.

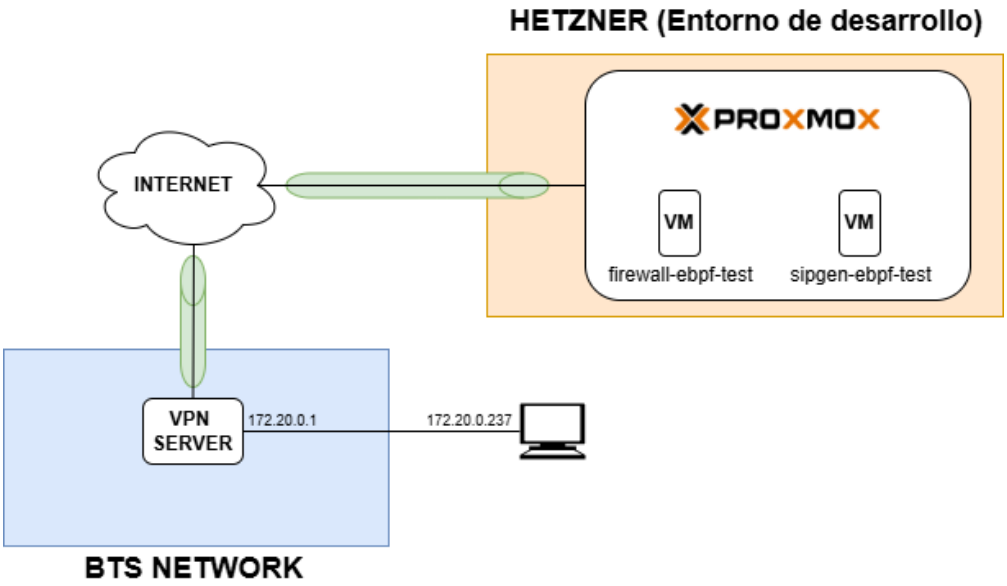


Figura 17. Conexión con el entorno de desarrollo en Hetzner.

En el entorno de pruebas que se tiene en Hetzner, se decide configurar **dos máquinas virtuales** para desarrollar el proyecto de forma que simule lo más fielmente posible su futura puesta en producción. Estas dos máquinas virtuales se alojan en el mismo centro de datos, lo que garantiza una conectividad completa entre ellas. Cada máquina virtual cumple una función específica, tal como se detalla en la *Figura 18.*

- **firewall-ebpf-test:** Esta máquina virtual se encarga de emular el funcionamiento de un "Dedge". Es el equipo donde **se configura y ejecuta el Firewall eBPF (agente eBPF)**, recibiendo tráfico SIP de prueba para evaluar la efectividad del sistema. Su función principal es permitir un análisis detallado de las capacidades del proyecto en un entorno controlado.

- **sipgen-ebpf-test:** La segunda máquina está destinada a generar tráfico SIP simulado, recreando escenarios reales de comunicación provenientes de Internet. Además de dirigir este tráfico hacia la máquina de prueba, también actúa como "**eBPF Master**". En ella se aloja el servidor web encargado de la gestión centralizada de los distintos agentes, proporcionando una interfaz de administración accesible y funcional.

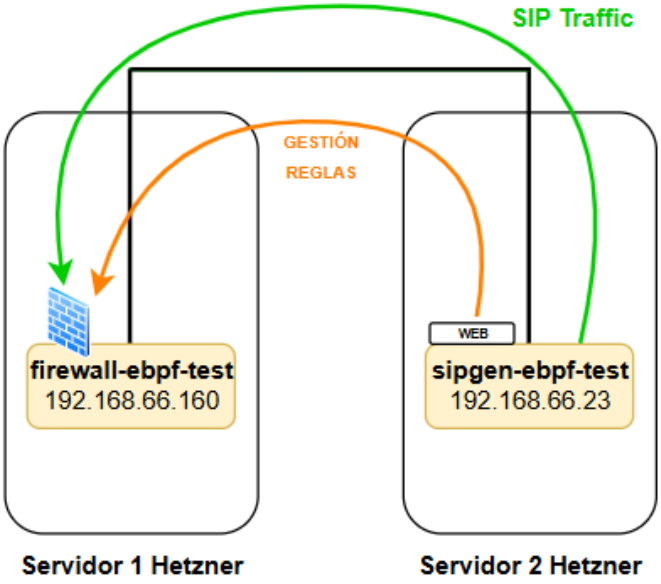


Figura 18. Máquinas virtuales y su funcionalidad.

Con esta configuración de pruebas, es posible emular de manera sencilla el escenario real en el que una máquina con el Firewall eBPF recibe tráfico SIP. Este entorno permite recrear situaciones similares a las que se presentan en producción, facilitando el análisis y evaluación del sistema en condiciones controladas. El escenario en un entorno de producción sería similar al siguiente:

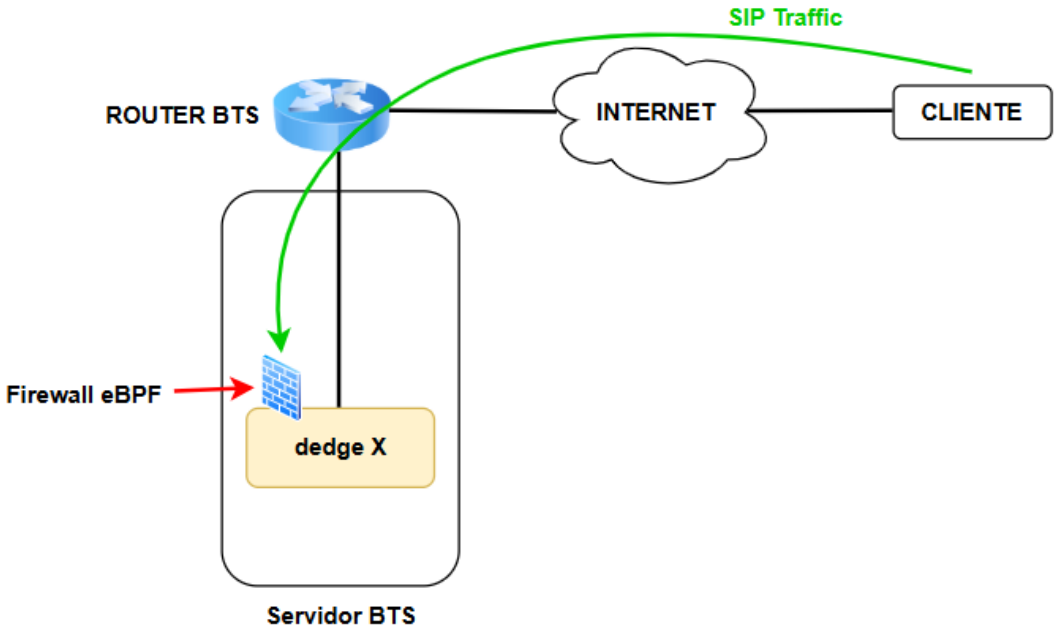


Figura 19. Escenario a simular en el entorno de pruebas.

Para el escenario de pruebas, se elige Debian 12 como sistema operativo en ambas máquinas virtuales. Esta elección se debe a que los equipos "Dedge", donde se implementaría el nuevo firewall, operan con este mismo sistema, garantizando así la total compatibilidad del desarrollo. En cuanto a los recursos asignados, se opta por configuraciones mínimas, dado que el proyecto no requiere inicialmente un consumo significativo de recursos.

3.3. Tecnologías aplicadas

En este apartado, se describirá el uso de las tecnologías que han sido clave en el desarrollo del proyecto, con el objetivo de proporcionar una base introductoria que sirva de apoyo a la explicación técnica detallada que se presentará posteriormente.

El enfoque principal del proyecto reside en el desarrollo desde cero de un **programa eBPF** que funcione como firewall. Los programas eBPF están diseñados para ejecutarse en el espacio del *kernel*, aprovechando la capacidad de realizar operaciones directamente sobre el tráfico de red con gran eficiencia. Para este proyecto, el programa se ha escrito utilizando la herramienta especializada **BCC (BPF Compiler Collection),** que proporciona una interfaz para el desarrollo y la implementación de programas eBPF, además de bibliotecas que simplifican la integración con lenguajes de alto nivel como Python.

Una de las tecnologías clave utilizadas es **XDP (eXpress Data Path),** una extensión de eBPF diseñada para procesar paquetes de red con una latencia extremadamente baja. XDP permite ejecutar programas eBPF directamente en el nivel más bajo de la pila de red, antes de que los paquetes sean procesados por el *kernel*. Esto lo convierte en una solución ideal para tareas de filtrado y manipulación de tráfico en tiempo real, como es el caso del Firewall eBPF que se desarrolla en este proyecto.

Gracias a XDP, el programa puede tomar decisiones inmediatas sobre si **permitir, bloquear o redirigir un paquete**, logrando un rendimiento superior al de enfoques tradicionales como *iptables*. Además, su integración con eBPF permite gestionar reglas de acceso dinámicas y altamente personalizables, lo que proporciona una gran flexibilidad para adaptarse a los requerimientos de un entorno de telecomunicaciones de alta demanda [11]. El esquema de la *Figura 20* muestra el procesamiento de un paquete aplicando XDP y eBPF

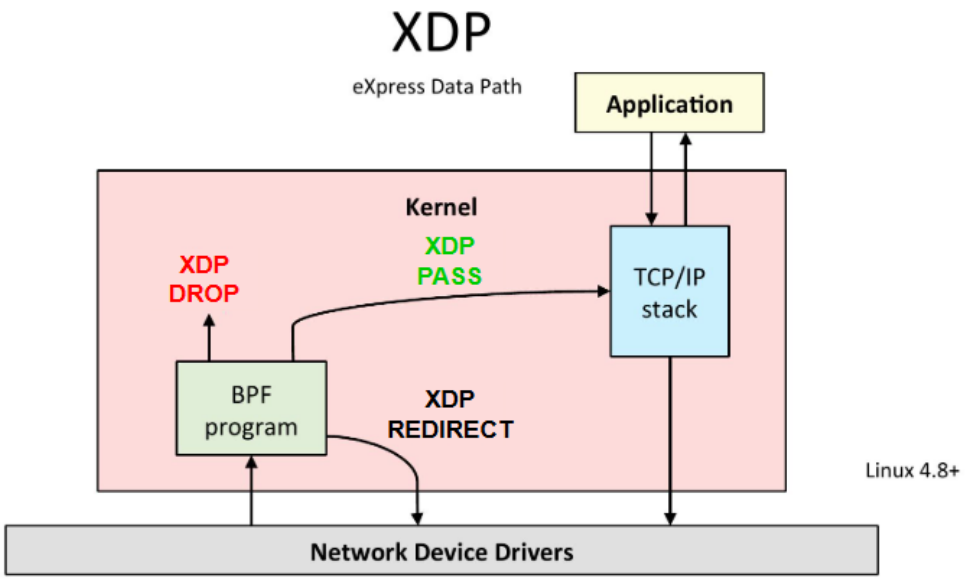


Figura 20. Funcionamiento interno de XDP.

El programa principal del sistema está desarrollado en **Python**, un lenguaje elegido por su flexibilidad, simplicidad y amplia disponibilidad de bibliotecas, especialmente para la **integración con eBPF**. Este programa actúa como el núcleo del sistema, siendo el encargado de cargar y gestionar el programa eBPF en el *kernel* del sistema operativo. Además, Python permite la interacción fluida con las herramientas de eBPF, como BCC.

En este contexto, Python se utiliza para definir las reglas del firewall, monitorear el tráfico procesado por el programa eBPF y establecer una interfaz de comunicación con otros componentes del sistema, como la API REST para la gestión centralizada. Esto convierte al programa principal en un **enlace clave** entre la lógica de filtrado de bajo nivel proporcionada por eBPF y las interfaces de alto nivel que interactúan con los usuarios y administradores del sistema.

Otra tecnología utilizada en el desarrollo es **Flask**. Se trata de un micro-framework para Python que se ha utilizado para desarrollar la interfaz web y las APIs necesarias para interactuar con el Firewall eBPF. En este caso, Flask se encarga de crear una serie de **endpoints** que permiten la gestión remota y dinámica del Firewall eBPF. Los administradores pueden interactuar con el sistema a través de una interfaz web, enviando solicitudes HTTP que son procesadas por las rutas definidas en el servidor Flask. Cada endpoint está diseñado para ejecutar acciones específicas, como agregar o eliminar reglas. Entre ellos, se intercambian datos que generalmente están en formato **JSON**.

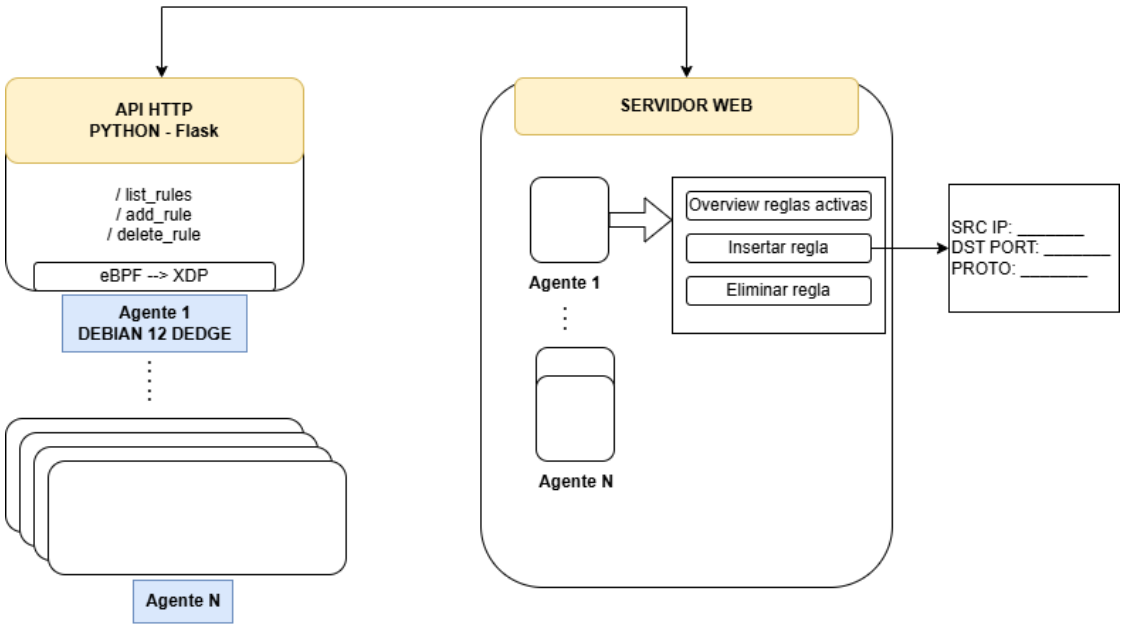


Figura 21. Conjunto de funcionalidades del sistema.

Otro lenguaje que se ha trabajado en el Firewall eBPF es **HTML.** El uso de HTML en este proyecto se centra en la creación de la interfaz web que interactúa con el Firewall eBPF, permitiendo a los administradores gestionar las reglas de tráfico de manera visual. HTML es el lenguaje de marcado que estructura el contenido de la página, facilitando la creación de formularios, tablas y botones que permiten a los usuarios agregar, eliminar y visualizar reglas en tiempo real. La interacción con el backend se realiza mediante llamadas **JavaScript** que comunican el navegador con el servidor Flask, utilizando los endpoints de la API.

En cuanto a la seguridad del Firewall eBPF, se ha trabajado con varias aplicaciones o tecnologías. Una de ellas es **PostgreSQL**, un famoso sistema de gestión de bases de datos relacional. Se ha aplicado tanto al sistema de *Login* como a la creación de la *Whitelist SIP*. La metodología utilizada así como las tablas y datos trabajados con PostgreSQL se detallan en el ANEXO C.

Otro punto a destacar es la implementación de **certificados TLS (Transport Layer Security)** en la comunicación tanto con la interfaz web como entre las APIs que interactúan en el sistema. Esto permite establecer una conexión segura mediante el **protocolo HTTPS**, proporcionando una capa adicional de protección contra ataques como el *man-in-the-middle*, donde los atacantes intentan interceptar y alterar la comunicación [12]. Las **peticiones HTTPS** son fundamentales en la comunicación segura entre cliente y servidor. En este proyecto, se utilizan principalmente dos métodos: **GET** y **POST**. Las solicitudes GET permiten obtener información del servidor, como las reglas activas del Firewall eBPF, mientras que las solicitudes POST se emplean para enviar datos, como la creación o eliminación de reglas. Todas las respuestas del servidor se envían en formato **JSON**, un estándar ligero y estructurado que facilita la interpretación y manipulación de los datos tanto en el cliente como en el backend. Esta implementación asegura la integridad y confidencialidad del tráfico web y las interacciones entre las APIs del Firewall eBPF.

3.4. Sistema completo

Una vez desarrolladas todas las funcionalidades y características que integrarán el Firewall eBPF, es posible obtener una visión general del sistema completo que se implementará. Este enfoque permite comprender el proceso global que implica la puesta en funcionamiento y explotación de nuestro sistema, destacando cómo interactúan sus diferentes componentes para alcanzar los objetivos planteados.

El sistema del Firewall eBPF se compone de varios elementos interconectados que trabajan en conjunto para garantizar su funcionalidad y eficiencia. En primer lugar, está el **núcleo del programa eBPF**, que actúa como firewall en el proceso de gestión del tráfico por parte de los equipos de la empresa, optimizando el manejo del tráfico SIP. Este núcleo se complementa con un servidor desarrollado en Python y Flask, que expone APIs para la configuración dinámica del firewall mediante *endpoints*. Su interacción resumida se puede observar en la *Figura 22*.

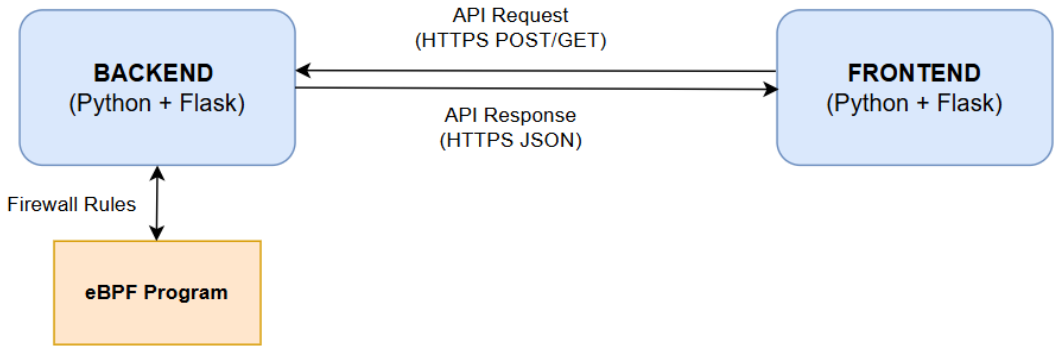


Figura 22. Interacción Backend – Frontend.

Además de las funcionalidades del *backend*, el sistema cuenta con una **interfaz web interactiva** diseñada en HTML y enriquecida con JavaScript, que permite a los usuarios gestionar de forma sencilla las reglas del Firewall eBPF. Esta interacción se realiza a través de una API que conecta la interfaz con el backend, facilitando la comunicación bidireccional. Esto queda descrito en la *Figura 23*.

Cuando el usuario realiza una acción, como listar las reglas activas o añadir una nueva, JavaScript envía una solicitud HTTPS a la API. Estas solicitudes permiten recuperar información del backend o enviar datos para su procesamiento. Por ejemplo, al listar las reglas, el backend responde con un JSON que contiene todas las reglas configuradas, y JavaScript utiliza esta respuesta para actualizar dinámicamente la tabla en la interfaz. De manera similar, al añadir una regla, el formulario envía los datos al backend, y la respuesta confirma si la operación fue exitosa, reflejando los cambios de inmediato en la página.

Esta arquitectura permite una experiencia de usuario fluida y en tiempo real, combinando el diseño estructurado del HTML con el comportamiento dinámico de JavaScript para garantizar una gestión efectiva y sencilla del firewall.

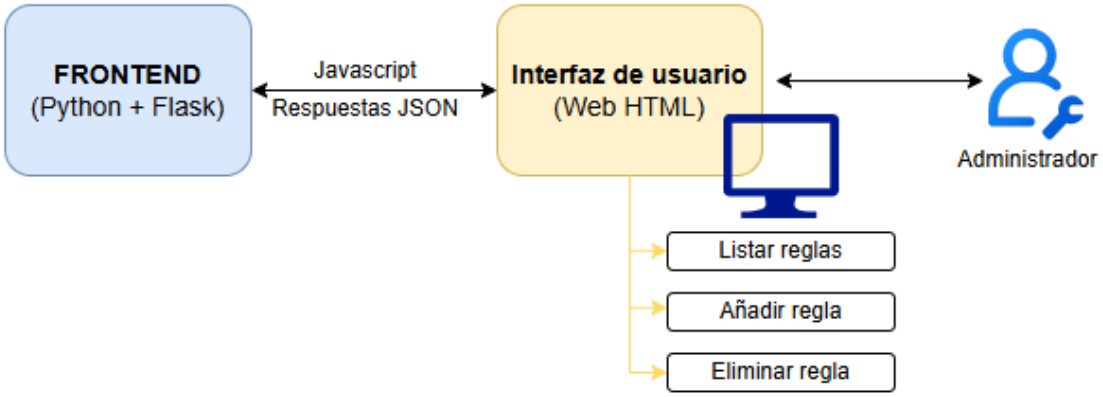


Figura 23. Interacción Frontend – Web HTML.

Tras haber descrito de forma general las dos partes principales del proyecto, es posible elaborar un esquema funcional que refleje el funcionamiento completo del sistema. Este esquema, representado en la *Figura 24*, proporciona una visión clara de su operación global.

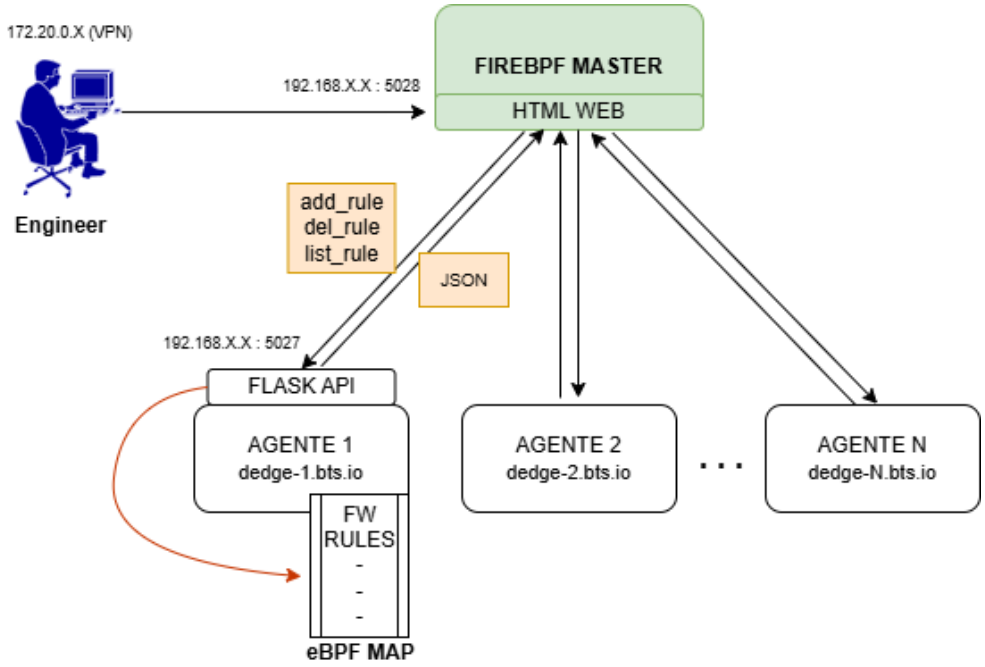


Figura 24. Operación global del sistema.

3.5. Instalación y configuración de herramientas a utilizar

Este apartado está dedicado a mostrar el proceso seguido para la instalación y configuración de las distintas herramientas o sistemas utilizados, así como del propio entorno de pruebas preparado.

- Entorno de pruebas

Como se ha explicado en el *Apartado 3.2*, el desarrollo y las pruebas del programa se llevan a cabo en un entorno virtualizado con máquinas nuevas, lo que convierte al proceso de instalación de software desde cero en un elemento fundamental de este proyecto.

La preparación de los equipos utilizados en el entorno de pruebas se puede consultar en detalle en el ANEXO D.

Una vez tenemos el sistema operativo listo y la red funcionando, podemos pasar a instalar todo lo necesario para el desarrollo del firewall, la gestión de archivos y las pruebas de funcionamiento.

- Instalación de software necesario

Esta sección está dedicada a la **instalación de todos los paquetes o dependencias** necesarias dentro de las máquinas virtuales con las que se trabaja, para el correcto funcionamiento de eBPF y demás programas.

El proceso de instalación concreto de los paquetes para cada herramienta y los comandos necesarios para ello pueden consultarse en el ANEXO D.

eBPF

Para que la máquina Debian 12 pueda trabajar con programas eBPF, es necesaria la instalación de una serie de herramientas y librerías esenciales, como *clang*, *elfutils-libelf-devel*, *gcc*, *bpftool*, *kernel-devel* y *libbpf*.

BCC

Para utilizar BCC (BPF Compiler Collection) como lenguaje para escribir y ejecutar programas eBPF, es necesario instalar los paquetes esenciales, como *bpfcc-tools* y *libbpfcc-dev*.

XDP

La aplicación de XDP a programas eBPF requiere la instalación de algunos paquetes como *libbpf* o *bpftool*. Además, para adjuntar programas con XDP directamente a interfaces de red, necesitamos la herramienta *iproute*.

Python

Para escribir y ejecutar programas en Python que interactúen con eBPF, se deben instalar ciertas herramientas, bibliotecas y dependencias como *python3*, *libbcc*, *jsonschema*.

Flask
Para implementar el backend en Python con Flask, se necesita la dependencia *flask*.

PostgreSQL, SSL
En el proceso de aplicación de seguridad al programa, se utiliza PostgreSQL y certificados SSL. Para todo ello, es necesario una serie de paquetes que permiten gestionar las dos herramientas, como *postgresql*, *postgresql-contrib*, *psycopg2*, *bcrypt*, *libssl-dev*.

SIPp
Para las pruebas de funcionamiento, se instala la aplicación de simulación de tráfico SIP llamada SIPp.

Capítulo 4: Desarrollo técnico completo del Firewall Linux

El desarrollo del firewall basado en eBPF abarca múltiples áreas de diseño, desarrollo e implementación que, aunque son independientes en su concepción, interactúan estrechamente para conformar un sistema integral. Dada la diversidad de tecnologías, herramientas y programas empleados para lograr la implementación completa, este apartado tiene como objetivo proporcionar una explicación detallada de cada una de ellas, así como de la forma en que se integran para materializar las funcionalidades descritas a lo largo de esta memoria.

La integración de las distintas funcionalidades del proyecto requiere, como paso inicial, una planificación cuidadosa para coordinar la interacción entre los diversos componentes del sistema. Para ello, se establece una estructura funcional general que organiza los procesos principales. Esta estructura se centra en programas que actúan como backend y frontend, cuya operativa se detalla a continuación.

- Backend: ***add_rule.py*** y ***app.py***
- Frontend: ***index.html***

Objetivo de funcionamiento general

El proyecto se compone de varias partes que trabajan juntas para proporcionar una funcionalidad completa. Por un lado, tenemos el archivo `**app.py**`, que es el corazón del **backend** de la aplicación. Este archivo abre un puerto en el servidor web (en este caso, el puerto 5028 – elegido aleatoriamente) utilizando un framework como Flask. Esto permite que los usuarios (los trabajadores de BTS) se conecten a la web de gestión del firewall desde sus navegadores. Cuando alguien accede al servidor web, `app.py` responde sirviendo el archivo `**index.html**`, que actúa como la interfaz gráfica visible para los usuarios.

El archivo `index.html` es el **frontend** del proyecto. Es lo que el usuario final ve y con lo que interactúa directamente. Contiene elementos como formularios, botones y tablas que permite realizar acciones como añadir, eliminar o listar reglas del firewall. Estas interacciones generan peticiones que se envían de vuelta a `app.py` para ser procesadas. Por ejemplo, si un usuario completa un formulario y lo envía para añadir una regla, el navegador envía esos datos al backend a través de una ruta específica.

Cuando `app.py` recibe esas peticiones, su función es interpretarlas y, si es necesario, delegar las tareas a otros componentes del backend, como `**add_rule.py**`. Este último es un script especializado que interactúa directamente con el sistema operativo y el programa eBPF. Se encarga de tareas específicas, como añadir, listar o eliminar reglas en el firewall utilizando herramientas como eBPF o XDP. Así, `add_rule.py` realiza las acciones técnicas sobre el firewall en sí mientras que `app.py` actúa como mediador, asegurándose de que todo fluya entre el usuario, el frontend y el sistema operativo.

El proyecto, por tanto, se divide en dos grandes partes: el backend y el frontend. El **backend** es la lógica detrás de las escenas; incluye tanto `app.py`, que gestiona las peticiones y responde

a los usuarios, como `add_rule.py`, que ejecuta acciones directamente en el sistema. Por otro lado, el **frontend** se encarga de la presentación y la interacción con el usuario, con `index.html` como su pieza central. Aunque `app.py` es técnicamente parte del backend, también desempeña un papel crucial al servir el frontend y proporcionar una forma para que el navegador del usuario interactúe con la lógica del sistema.

A partir de este punto, se va a desarrollar cada una de las partes que componen el sistema, analizando a fondo su papel en la implementación final. Las partes de código mencionadas explícitamente pueden consultarse en el ANEXO E.

El código completo con todas sus partes queda subido en un repositorio público de *Github* para su consulta: https://github.com/cirigoyen277/firewall_ebpf_cirigoyen

4.1. Programa eBPF para el análisis de paquetes

El desarrollo del programa eBPF constituye el núcleo central de este proyecto, ya que es responsable de la función más crítica en términos de rendimiento: analizar los paquetes de red tan pronto como llegan a la interfaz de red del equipo.

La programación de este componente se ha realizado en el **lenguaje C** debido a las restricciones del *kernel* de Linux, utilizando además **BCC**. Como se ha detallado en el *Apartado 3.3*, BCC simplifica significativamente tanto el desarrollo como el uso de programas eBPF. Además, el programa se adhiere a la interfaz de red mediante **XDP**, lo que permite el análisis inmediato de los paquetes.

El propósito del programa es claro y esencialmente similar al de cualquier firewall: analizar los paquetes de red. En un sistema como este, que gestiona reglas de permiso y denegación, es fundamental consultar una especie de "base de datos" para determinar si un paquete debe ser permitido o rechazado. En eBPF, esta función se logra mediante los Mapas eBPF, cuya explicación puede consultarse en el *Apartado 2.2.1*.

De este modo, como se describe en la *Figura* 25, cuando un paquete alcanza la interfaz de red, el programa eBPF consulta en los Mapas eBPF si el tipo de tráfico está permitido según las reglas configuradas y, utilizando funciones XDP, toma acción sobre él. Estas funciones permiten bloquear, aceptar o redirigir el tráfico de manera eficiente. Un esquema resumido del funcionamiento puede consultarse en la *Figura 25*.

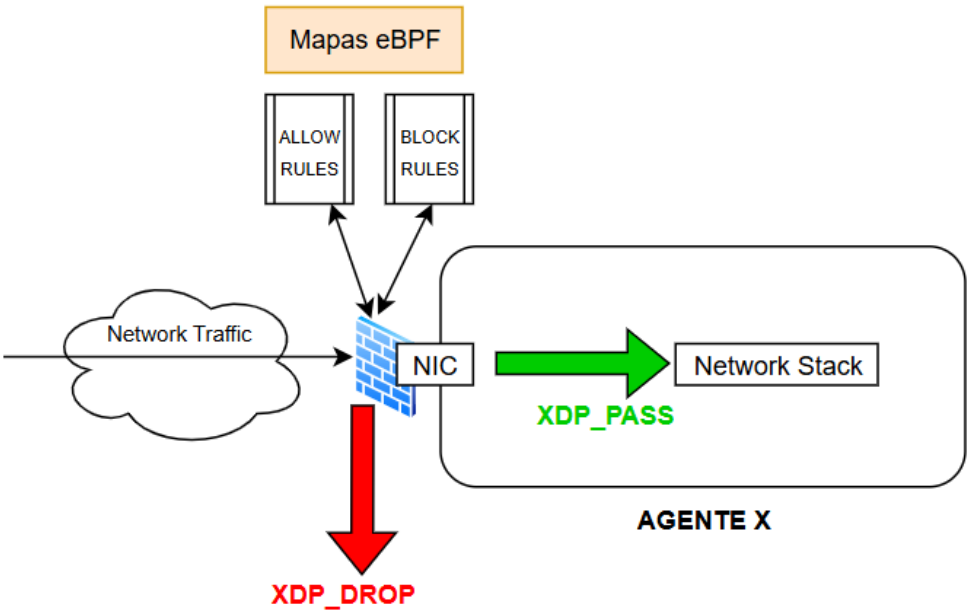


Figura 25. Consultas del Firewall eBPF a los mapas.

El programa eBPF se invoca desde el script de Python 'add_rule.py', y su código está "embebido" dentro del propio archivo. Esto significa que el script Python incluye el código en lenguaje C del programa eBPF, que se compila y carga dinámicamente en el *kernel* de Linux desde el mismo 'add_rule.py'. A continuación, se repasan una a una las partes del programa.

Las partes de código referentes a lo mencionado a lo largo de este apartado pueden consultarse en el ANEXO E – Sección 1

Inicialmente, se incluyen estructuras y definiciones esenciales necesarias para analizar y procesar paquetes de red en el nivel del *kernel*.

Posteriormente, se definen unas estructuras de datos. Estas definen los tres casos de parámetros de red que se van a analizar para tomar decisiones sobre los paquetes de red. En el Firewall eBPF, se puede trabajar con reglas que analicen:

- Protocolo de los paquetes. (*rule_key_proto)*
- IP Fuente de los paquetes. (*rule_key_ipsrc*)
- Protocolo, IP Fuente y Puerto Destino de los paquetes. (*rule_key_ipsrc_proto_portdst*)

Por tanto, se generan tres estructuras de datos distintas que se rellenan con los datos de los paquetes de red. Al llegar un paquete, se generan las tres estructuras con sus datos, y se revisan las reglas del firewall para ver si en algún punto alguna de ellas hace "match" y por tanto se tomará una decisión XDP.

Al definir las estructuras se define el tipo de datos que las conformarán.
Protocolo (proto): Entero sin signo de 8 bits (los protocolos tienen identificadores numéricos).
IP Fuente (ip_src): Entero sin signo de 32 bits (dirección Ipv4 estándar).
Puerto destino (port_dst): Entero sin signo de 16 bits.

Como se ha comentado, las reglas del firewall se almacenan en **Mapas eBPF**. Al trabajar con tres estructuras distintas a revisar (las que se acaban de presentar), se generan 3 mapas eBPF – tanto para permitir como para bloquear - distintos que almacenan cada tipo de estructura. Así, las reglas se generarán como "estructuras".

Por ejemplo, si se quiere bloquear todo el tráfico proveniente de la IP "1.1.1.1", se guardará una entrada en el mapa "blocked_rules2" con una estructura tipo *rule_key_ipsrc* que contenga la IP "1.1.1.1". Cuando llegue un paquete desde esa IP, se buscará su *rule_key_ipsrc* en los mapas, y al hacer "match", se bloqueará.

Los mapas **BPF_HASH** son una de las estructuras de datos más utilizadas en eBPF, diseñados para almacenar pares clave-valor de manera eficiente en el *kernel* de Linux. Aunque existen más tipos de Mapas eBPF, se ha elegido este por sus funcionalidades para buscar, añadir o eliminar datos [13].

En este punto, comienza el desarrollo de la propia función que toma decisiones sobre los paquetes con XDP. Se ha llamado *block_packet().*

En primer lugar, se realiza una verificación del tipo de paquete que se analiza, con el objetivo de evitar accesos inválidos en memoria, lo que podría suponer riesgos importantes para la seguridad del sistema.

Ahora, se procede a extraer la información de los paquetes de red analizados. Se van a generar los tres tipos de estructura con los que se trabaja, de la siguiente forma:

- Extracción del **protocolo** de red: *ip->protocol*
- Extracción de la **IP fuente** del paquete: *ip->saddr*
- Extracción del **puerto destino** del paquete: *tcp->dst*

Una vez obtenidos estos datos, se generan las tres estructuras ya vistas con los nombres *key1, key2* y *key3*.

Con las "key" ya listas, podemos pasar a verificar las reglas del firewall para actuar sobre el paquete. Los Mapas eBPF permiten comprobar si una esturctura hace "match" con alguna de sus entradas directamente con una función *lookup*. De esta manera, el proceso es sencillo y efectivo.

El proceso de decisión sobre los paquetes se basa en revisar primero los mapas con las reglas de PERMITIR el paso (*allowed_rulesX*). Si hace "match" con alguna de las reglas, el resto de los mapas no son revisados y el paquete pasa al stack de red directamente mediante un XDP_PASS. Si no, el programa pasa a evaluar las reglas de BLOQUEAR el paso (*blocked_rulesX)*, y si encuentra coincidencia con alguna de las estructuras del paquete, hace el XDP_DROP. Por defecto, la política del Firewall eBPF es no aceptar el tráfico, por lo que,

si no hace "match" con nada, cualquier tráfico se bloquea (mismo funcionamiento que el *iptables*). El esquema del funcionamiento puede consultarse en la siguiente *Figura 26*.

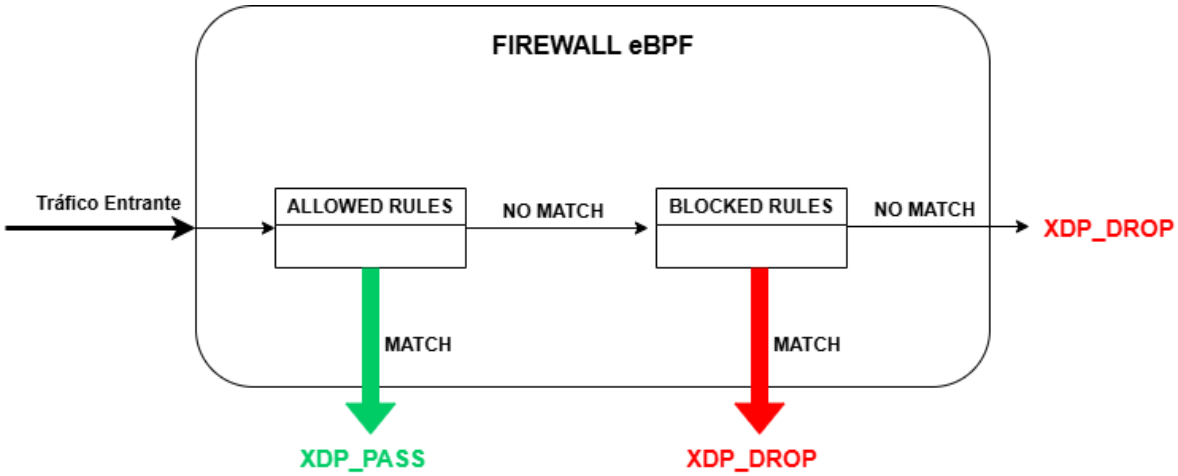


Figura 26. Paso del tráfico por los Mapas eBPF.

Además, los mapas se evalúan según su **nivel de concreción**. Es decir, primero deben analizarse las reglas más "generales". Si un paquete proveniente de la IP 1.1.1.1 llega al equipo, no tiene sentido analizar su IP, puerto y protocolo antes que solamente su IP, ya que podría haber una regla permitiendo todo el tráfico desde la 1.1.1.1 y nos ahorraríamos evaluar el resto de los mapas.

Por ello, tanto para los mapas de permitir tráfico como para los de bloquearlo, primero se evalúa el que contiene sólo el protocolo, después sólo la IP origen, y en último lugar el que contiene ambas junto al puerto destino. Para ello, se genera para cada "key" una variable *allow_valueX* o *block_valueX* que funciona como un dato boolean y que obtendrá valor si la función de *lookup* en el mapa concreto tiene éxito, es decir, si hace "match" con alguna regla existente:
*u32 *allow_value1 = allowed_rules1.lookup(&key1);*

Si hace "match" con una regla de *allowed_rulesX*, se aplica **XDP_PASS**. Si lo hace con una de *blocked_rulesX* se aplica **XDP_DROP** y se tira el paquete. Y como se ha comentado, si no hace "match" con nada, por defecto se aplica un **XDP_DROP** al final.

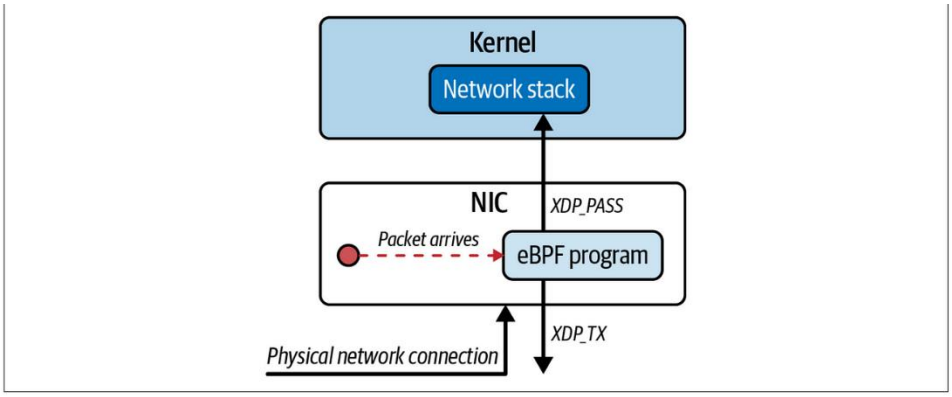


Figura 27. Esquema de aplicación de XDP.

4.2. Programa backend para los agentes: “add_rule.py”

El programa *add_rule.py* se instala en los **agentes eBPF** en los que se aplica el firewall y está escrito en **Python**. Actúa como una interfaz entre el espacio de usuario y el programa eBPF, cargado en el *kernel*. Su función principal es gestionar las reglas del firewall dinámicamente a través de una **API REST** implementada con **Flask**. Este script interactúa con los mapas eBPF (BPF_HASH) para añadir, listar y eliminar reglas que bloquean o permiten tráfico según los parámetros ya mencionados.

Las reglas se cargan directamente en los mapas asociados (*allowed_rules* y *blocked_rules*), y el programa eBPF toma decisiones basadas en estas reglas para manipular el tráfico en tiempo real. Además, *add_rule.py* incluye funciones para leer una *whitelist* desde una base de datos y limpiar el entorno al salir. En resumen, este script permite **modificar el comportamiento del programa eBPF de manera dinámica sin reiniciar el sistema**. A continuación, se explica su funcionamiento sección por sección.

Las partes de código referentes a lo mencionado a lo largo de este apartado pueden consultarse en el ANEXO E – Sección 2

Inicialmente, en el programa se incluyen las bibliotecas que emplea el script, como *Flask* para crear la API REST, *bcc.BPF* para cargar y gestionar el programa eBPF, y *socket, struct, ctypes* para manipular IPs, estructuras y tipos en C. Además, se define la interfaz de red del equipo donde actuará el firewall, en este caso “enp6s18”. También se inicializa la aplicación Flask en Python, dándole el nombre de “app”.

Seguidamente, se escribe el programa eBPF (embebido) que se ha visto en el apartado anterior, y se le da el nombre de “**bpf_program**”. Éste se carga utilizando su nombre, y utilizando la variable “b”, se carga también la función interna del propio programa eBPF “*block_packet()*” en el *kernel*. Además se indica que se utilizará XDP.
En la siguiente línea, vinculamos el programa eBPF cargado a la interfaz de red ya definida, usando XDP.

Posteriormente se obtienen los mapas HASH eBPF, a partir del propio programa con *b.get_table().* De igual forma para los 6 que se habían creado.

Se programa también una función (*cleanup(sig, frame)*) a la que se llama al cortar el programa Python, para así desactivar eBPF en la interfaz de red.

Es necesaria también una función (*ip_to_u32_inverted(ip)*) para pasar las **IPs a bytes** e invertirlos. Esto es debido a que las IPs en los mapas eBPF se leen en sentido contrario al habitual. Por ello, si queremos bloquear una IP, debemos almacenarla en su forma inversa. La siguiente función convierte una dirección IP en formato String a un número de 32 bits con inversión.

A continuación, inicia la sección del código responsable de cargar las reglas predeterminadas al ejecutar el programa. Se han definido tres tipos de reglas que pueden añadirse automáticamente al inicio:

- ***Whitelist* para acceso completo**. Estas reglas leen una *whitelist* de un archivo "whitelist_all.txt" modificable, y otorgan **acceso completo a la máquina** desde las IPs que la componen. En esta *whitelist*, por ejemplo, podríamos añadir la IP del eBPF Master. Se añade una entrada por cada IP al mapa *allowed_rules2*.

- ***Whitelist* para tráfico SIP.** Este conjunto de reglas es esencial para el firewall de las máquinas BTS que manejan tráfico SIP. Actualmente, el funcionamiento basado en *iptables*, como se detalla en el *Apartado 2.3*, ya incluye esta funcionalidad. En nuestro firewall con eBPF se ha querido implementar de igual forma, **leyendo la *Whitelist* desde una base de datos PostgreSQL**. Esto permite una mayor escalabilidad y una gestión automatizada y eficiente de estas reglas.
  En el desarrollo del proyecto en el entorno de pruebas, la base de datos con las IPs que pueden pasar tráfico SIP (*whitelist*) se encuentra en un PostgreSQL de la misma máquina, que simula una base de datos externa como con la que cuenta BTS.

  Las IPs se almacenan en una tabla de PostgreSQL como la representada en la *Figura 28*. La secuencia de comandos utilizada para visualizar la tabla en concreto se puede consultar en el ANEXO C.

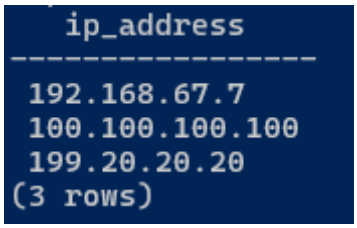


Figura 28. Tabla de PostgreSQL con las IPs a permitir.

  Estas IPs se leen mediante una función en Python (*generar_whitelist__sip())* integrada en el propio add_rule.py, que hace una Query a la base de datos para extraer todas las IPs y plasmarlas en un fichero de texto, llamado "*whitelist_sip.txt*".

  Una vez extraídas las IPs desde la base de datos al archivo de texto, una función similar a la de la *whitelist* general crea las reglas del firewall (en este caso en el mapa *allowed_rules3*, permitiendo conexiones desde esas IPs con el protocolo **UDP** y el puerto destino **5060** (puerto SIP).

- **Reglas por defecto personalizadas.** Otro aspecto incluido en el diseño es la capacidad de cargar reglas completamente personalizadas en el firewall al iniciar. Esta funcionalidad resulta especialmente útil, ya que permite configurar **reglas esenciales para el correcto funcionamiento y gestión inicial de la máquina**, como habilitar el

acceso SSH para el administrador. En este caso, las reglas vuelven a cargarse en el mapa *allowed_rules3* al especificarse los tres campos. El código muestra un claro ejemplo de acceso por SSH (puerto 22) para un usuario con IP 172.20.0.100.

A partir de este punto, donde ya se ha cargado el programa eBPF y se han definido las políticas de arranque, se pasan a definir los **endpoints** de la API REST con Flask. El programa trabaja con cuatro endpoints principales, ilustrados en la siguiente *Figura 29*, y cuyo código puede consultarse en el ANEXO E – Sección 2.

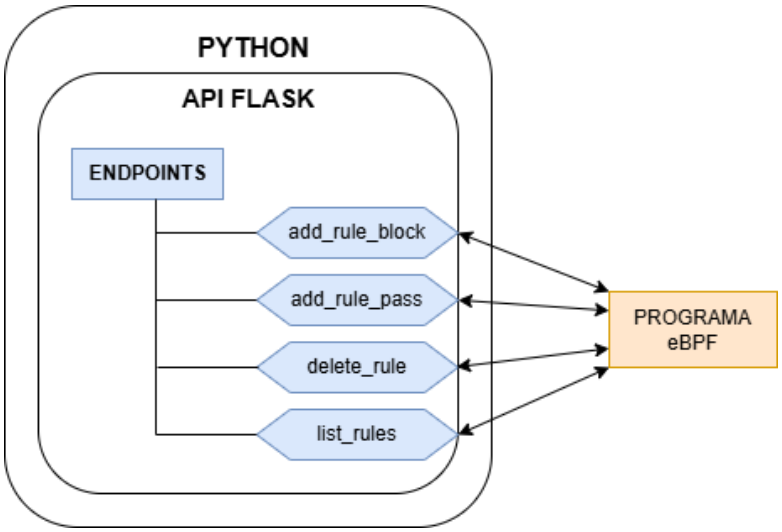


Figura 29. Esquema de interacción API - eBPF

- **/add_rule_block**:

Este endpoint permite añadir reglas al mapa eBPF para **bloquear tráfico** no deseado en función de tres criterios: protocolo, IP de origen, y puerto de destino. Los datos para añadir una regla a bloquear se reciben en una solicitud **POST** en formato JSON.

En consonancia con las estructuras de los mapas eBPF creados, el endpoint acepta JSON con sólo IP fuente, sólo protocolo y con ambos junto a puerto destino. Se analizan los datos recibidos en el endpoint, y se añade la regla al mapa correspondiente (*blocked_rulesX*). También se detectan datos inválidos o incompletos.

- **/add_rule_pass**:

Permite añadir reglas para **permitir tráfico**, y su funcionamiento es exactamente igual que para el */add_rule_block*: extracción de datos JSON recibidos en una solicitud **POST**, análisis de los mismos y adición de reglas (en este caso a los mapas *allowed_rulesX*).

- **/list_rules**:

El endpoint */list_rules* está preparado para recibir solicitudes **GET** y proporciona una **lista completa de las reglas activas** en el Firewall eBPF, tanto de bloqueo como de permiso. Consulta los mapas de reglas configurados (permitidas y bloqueadas), convierte las claves almacenadas en un formato legible (como direcciones IP y puertos) y **devuelve un JSON con**

todas las reglas activas clasificadas por acción ("block" o "allow"). Esto permite al usuario revisar fácilmente las configuraciones actuales del firewall.

- **/delete_rule**:

Este endpoint está diseñado para recibir solicitudes **POST** con datos JSON (protocolo, dirección IP de origen y puerto de destino), y permite eliminar reglas específicas del Firewall eBPF según los criterios proporcionados. Dependiendo de si la regla es de bloqueo o permiso, se identifica el mapa correspondiente y se hace una búsqueda de los parámetros recibidos. Al encontrar la entrada correspondiente, se elimina del mapa eBPF y la regla deja de tener efecto. Devuelve un mensaje confirmando la eliminación o un error si la regla no se encuentra o los datos son inválidos.

La parte final del código define el punto de entrada para ejecutar el programa Python e inicia el servidor Flask. Este escucha en el **puerto 5027** en todas sus direcciones IP.

4.3. Programa backend para el eBPF Master: "app.py"

El programa *app.py* se instala en el **eBPF Master** que gestiona el firewall de los distintos agentes y está escrito en **Python**. Implementa una aplicación web con Flask y actúa de **puente** entre la interfaz gráfica que ve el usuario y el programa eBPF en los agentes del firewall. Las solicitudes hacia el backend eBPF que se acaba de explicar se realizan mediante una **API protegida con SSL.** Además, el programa utiliza una base de datos PostgreSQL para **autenticar usuarios**, funcionalidad que se explicará más a fondo en el *Apartado 4.5.* Además, incluye otras medidas de seguridad como sesiones protegidas y el uso de **JSON Schemas** para verificar que el tipo de datos que se envían al backend es el esperado.

Las partes de código referentes a lo mencionado a lo largo de este apartado pueden consultarse en el <u>ANEXO E – Sección 3</u>

Inicialmente, se incluyen las bibliotecas y herramientas a utilizar a lo largo del programa.

Las primeras líneas de código están dedicadas a **definir los JSON Schemas**. Son estructuras que definen y validan el formato, tipos de datos y restricciones de los JSON, asegurando que cumplan con un conjunto específico de reglas. Para la interacción con el Firewall eBPF, se define:

- IP fuente – String en formato Ipv4
- Puerto destino – Entero en el rango 1-65535
- Protocolo – String entre: TCP, UDP, ICMP o "all"
- Acción – String: "block" o "allow"

Posteriormente se inicia la aplicación **Flask** en Python, y se define la **dirección de la API que trabaja con el programa eBPF** y sus reglas (el backend ubicado en el agente), siendo este ***API_BASE_URL = "https://<IP_Agente>:5027***.

Para el desarrollo del proyecto, la máquina que carga el firewall es *firewall-ebpf-test* (192.168.66.160) y el puerto en el que se ejecutó la API en el agente es el 5027:
API_BASE_URL = "https://192.168.66.160:5027"

A partir de aquí, se definen las funciones y endpoints necesarios para la gestión de los agentes eBPF desde la web. Inicialmente, se programan varias **funciones para el proceso de login** previo al manejo de las reglas del firewall, que se detallarán en el *Apartado 4.5*. Una vez el administrador del sistema se ha autenticado correctamente, es redirigido a la página principal de la web. El endpoint al que se accede es */login_ok*, y carga el archivo HTML que visualizaremos en el navegador.

En este punto se pasan a programar los endpoints encargados de **interactuar con el backend del agente eBPF** y su firewall para añadir, eliminar y listar reglas existentes.

- **/add_rule**

Este endpoint se encarga de enviar la directriz de añadir una regla al firewall del agente, tanto para permitir como para bloquear tráfico. Envía **peticiones POST al backend *add_rule.py*** instalado en el agente eBPF.

Si se quiere añadir una regla de permiso, se utiliza el endpoint **"*https://<IP_Agente>:5027/add_rule_pass*"**.
Si la regla a añadir es de bloqueo, se utiliza **"*https://<IP_Agente>:5027/add_rule_block*",**
Siendo *<IP_Agente>* la IP del agente cuyo firewall se quiere modificar.

Los datos para añadir una regla se reciben desde el formulario que el administrador rellena en la web, donde les da valor a los distintos parámetros. En el endpoint se extraen estos datos (acción, IP, puerto y protocolo), **recibidos en formato JSON**, y según la acción sea permitir o bloquear se decide qué endpoint del backend utilizar, como se describe en la *Figura 30*.

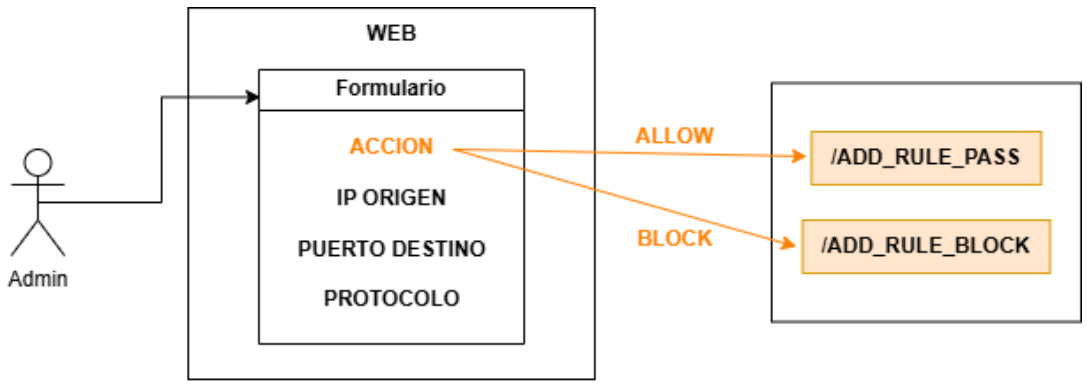


Figura 30. Elección del endpoint en función del formulario.

Una vez extraídos los datos, se forma un "payload" con ellos que se verifica frente al JSON Schema y se envía finalmente al backend utilizando el módulo *requests*:
endpoint = "/add_rule_block" if action == "block" else "/add_rule_pass"
requests.post(f"{API_BASE_URL}{endpoint}", json=payload, verify='cert_back.pem')

- **/delete_rule**

El endpoint */delete_rule* actúa prácticamente igual que el anterior. Extrae los datos recibidos en formato JSON de la regla a eliminar. Crea un "payload" con ellos, lo verifica frente al JSON Schema y si es correcto lo envía al endpoint del backend, mediante:
requests.post(f"{API_BASE_URL}/delete_rule", json=payload, verify='cert_back.pem')

- **/list_rules**

Este endpoint tiene la función de leer todas las reglas activas en el Firewall eBPF para mostrarlas en la web. Simplemente hace una petición **GET** al endpoint */list_rules* del backend del agente y recibe las reglas activas en formato JSON.
response = requests.get(f"{API_BASE_URL}/list_rules", verify='cert_back.pem')
rules = response.json()

En la última parte del código, se define un endpoint que es llamado desde la web y permite desplegar el "hostname" del agente eBPF gestionado en ese momento. Esto es realmente útil ya que desde el eBPF Master se manipulará más de un firewall.
Finalmente, se inicia el servidor Flask para la web en el **puerto 5028**:

4.4. Frontend en el eBPF Master: Web de gestión centralizada

La web de gestión centralizada para el programa eBPF proporciona una interfaz intuitiva basada en **HTML Y CSS** (*index.html y styles.css*). El archivo ***index.html*** define la estructura de la página, con una sección para mostrar las reglas activas en el agente, botones para eliminar reglas activas y el formulario para añadir nuevas reglas. El archivo ***styles.css*** estiliza la interfaz para mejorar la experiencia del usuario.

Esta web se comunica con el backend implementado en ***app.py***, que actúa como intermediario entre la interfaz web y el programa eBPF. Las peticiones realizadas desde la web, como añadir, eliminar o consultar reglas, son enviadas al backend mediante peticiones GET y POST. Allí se procesan y se transmiten al programa eBPF como ya hemos visto, para su aplicación al firewall. Este diseño separa el **frontend** y el **backend**, asegurando la gestión eficiente y accesibilidad al programa eBPF.

Para conectarnos a la web y gestionar los agentes, debemos acceder al puerto que hemos abierto con Flask en ***app.py*** en el eBPF Master (5028). Podemos acceder directamente desde nuestro navegador, mediante:

https://<IP_eBPF_Master>:5028/

siendo <*IP_eBPF_Master*> la IP del equipo que lleva la gestión centralizada (el Master). Una vez carguemos la URL en el navegador, seremos redirigidos a la **pantalla de login**, que tiene el siguiente aspecto (*Figura 31*):

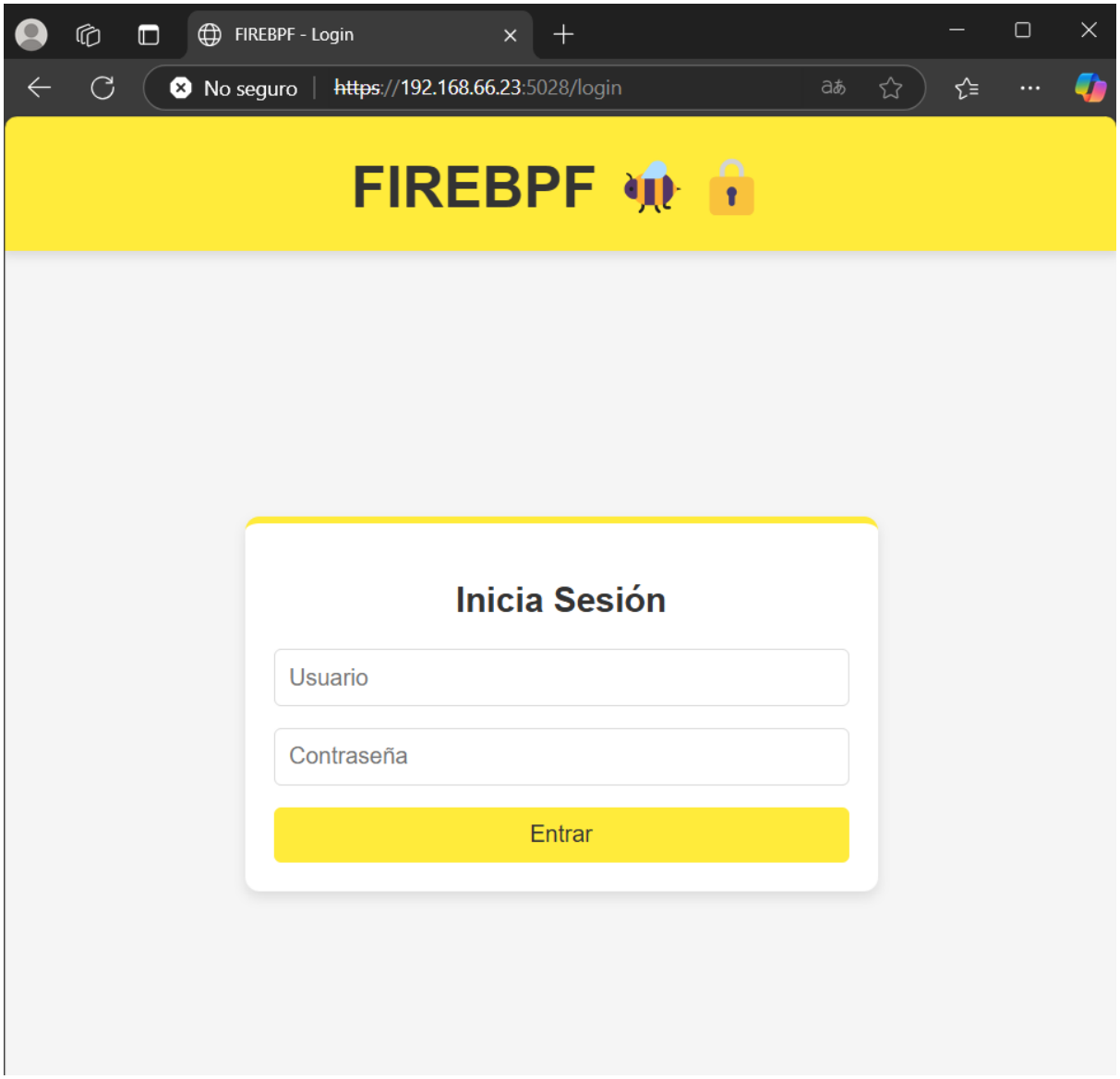


Figura 31. Ventana de Login en la web de gestión.

Una vez el administrador se identifica correctamente, se nos redirige a la página principal (*Figura 32*). En la parte superior, se reconoce el título "FIREBPF", y se identifica el agente al que estamos conectados. En el ejemplo para el proyecto, "*Agent: firewall-ebpf-test.bts.io*".

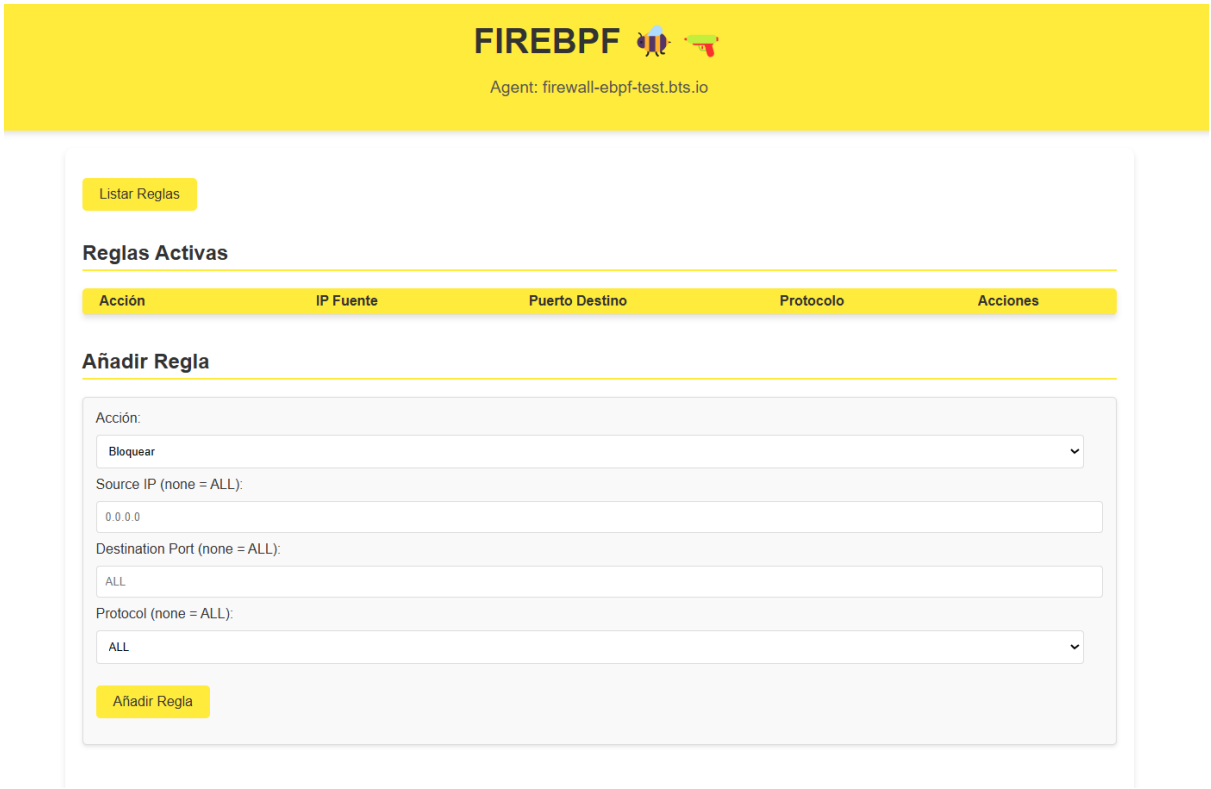


Figura 32. Página principal de la web.

Desde esta ventana, se pueden realizar todas las acciones relacionadas con la gestión de reglas. Al hacer clic en "**Listar Reglas**", se pueden ver por pantalla las reglas activas en el agente (*Figura 33*).

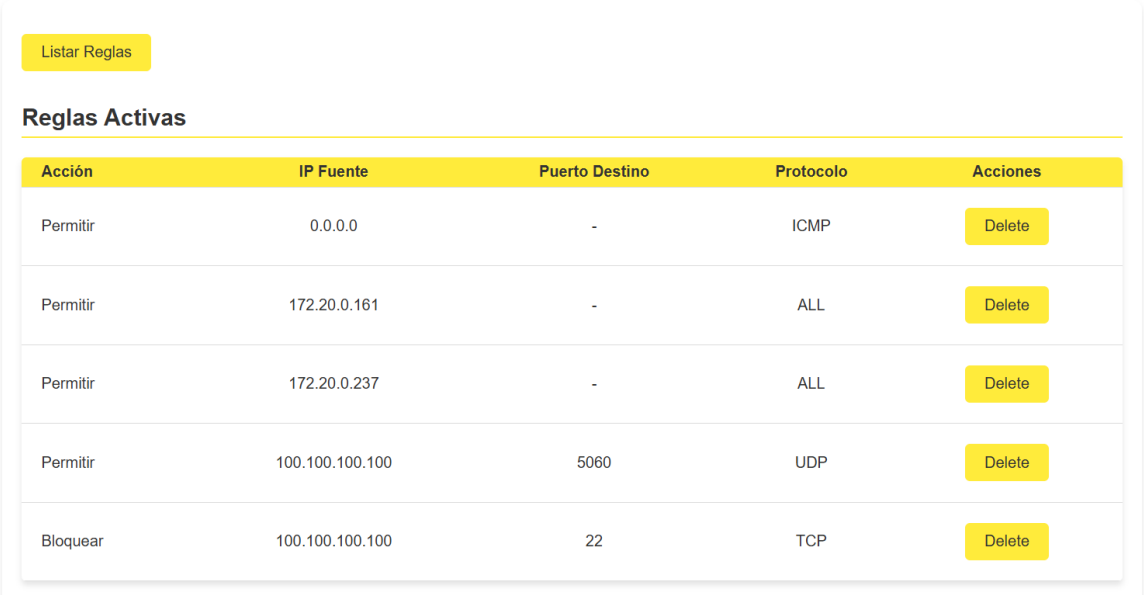


Figura 33. Visualización de reglas activas.

Es posible también eliminar cualquier regla, y dejará de tener efecto al momento en el agente sin ningún tipo de reinicio. Haciendo clic en **"Delete"** (margen derecho) para la regla que nos interese, se recibirá un mensaje en la misma web si se ha eliminado correctamente (*Figura 34*).

Figura 34. Mensaje de la web.

En la parte inferior de la web, se encuentra el formulario para añadir cualquier regla al Firewall eBPF del agente. Tras rellenar los campos que interesen y la acción a aplicar *(Figura 35)*, haciendo clic en **"Añadir Regla"** se verán inmediatamente en el listado de reglas activas, sin necesidad de ningún reinicio o recarga (*Figura 36)*.

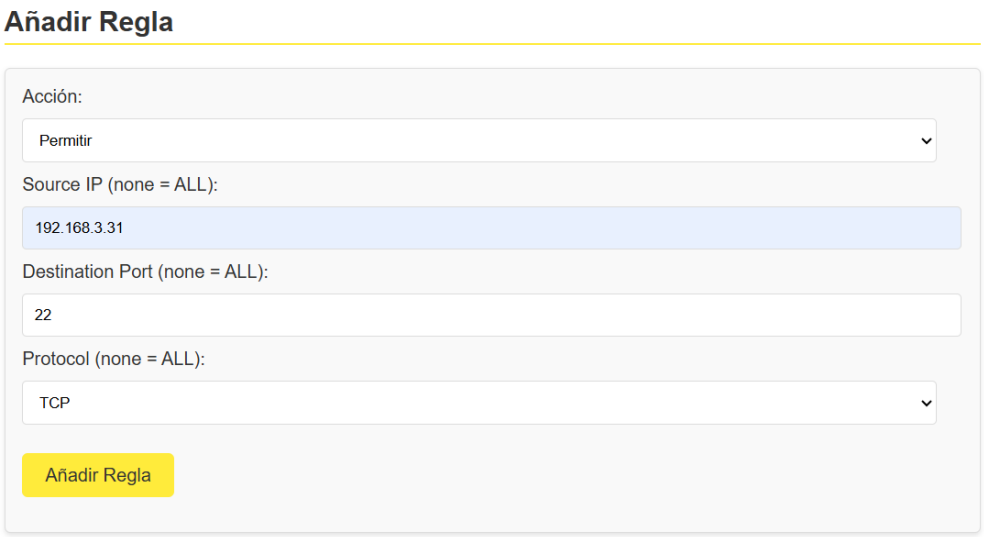


Figura 35. Formulario para añadir regla al Firewall.

Figura 36. Nueva regla aplicada.

Respecto al **código HTML (index.html)**, en resumen, utiliza funciones escritas en JavaScript para llamar a los endpoints de ***app.py*** al realizar acciones sobre la web, como presionar botones o rellenar formularios. Las respuestas recibidas, se plasman por pantalla de una manera clara y organizada.

Finalmente, la función del código **CSS (styles.css)** es preparar lo programado en HTML para darle un aspecto agradable y sencillo.

4.5. Aplicación de políticas de seguridad

Un aspecto crucial del Firewall eBPF desarrollado es la implementación de diversas políticas de seguridad, diseñadas para construir un **sistema robusto, completo y resistente**. Entre las medidas adoptadas destacan: un sistema de autenticación mediante **login** para la interfaz web de administración, el uso de **certificados TLS** para garantizar una comunicación HTTP segura, y la validación de datos a través de **JSON Schemas**. El uso y aplicación de cada una de estas medidas se detalla en el ANEXO F.

4.6. Puesta en marcha del sistema

Para poner en marcha el sistema completo, es necesario iniciar los programas diseñados para ambos puntos de la infraestructura: el eBPF Master y el/los agente/s. Este proceso se detalla en el ANEXO F, y su funcionamiento se resume gráficamente en la siguiente *Figura 37:*

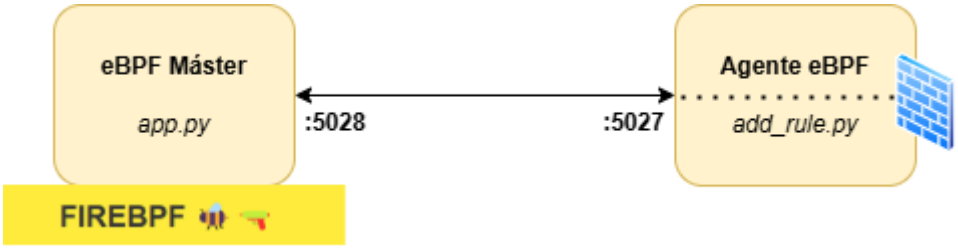


Figura 37. Comunicación entre equipos y puertos.

Capítulo 5: Pruebas de funcionamiento y resultados

Una vez que el firewall está configurado y operativo en los agentes eBPF permitiendo listar, añadir y eliminar reglas de paso y bloqueo, es momento de **evaluar su desempeño** en escenarios que reflejen las condiciones reales del entorno objetivo del proyecto.

Primero, se prueban diversas funcionalidades del Firewall eBPF, como la incorporación de usuarios al sistema, la carga de reglas predeterminadas y la lectura de IPs desde una base de datos para construir las *whitelist*s. Una vez verificadas estas capacidades, se procede a analizar su funcionamiento como firewall, comprobando su capacidad para bloquear y permitir tráfico específico.

Primero, se simulará el bloqueo de un tipo de tráfico específico. Luego, se realizarán pruebas con tráfico realista, simulando un entorno de alta carga donde el Firewall eBPF recibe grandes volúmenes de **tráfico SIP**. Se diseñarán distintas situaciones para evaluar el comportamiento del sistema en cada caso.

5.1. Pruebas iniciales

Las **pruebas iniciales**, referentes a las funcionalidades generales del firewall y bloqueo de tráfico concreto, pueden consultarse en el ANEXO G.

5.2. Pruebas realistas con tráfico SIP

Una vez verificada la capacidad del Firewall eBPF para gestionar todas sus funcionalidades y bloquear tráfico real que llega a la máquina, se procede a evaluar su desempeño en situaciones más representativas: el manejo de tráfico realista.

Los equipos donde irá instalado el programa eBPF cursan **grandes volúmenes de tráfico SIP**, que en ocasiones puntuales pueden sobrecargar los recursos, o no permitir su correcto aprovechamiento. Por ello será de gran relevancia evaluar la respuesta ante **situaciones realistas**, utilizando el Firewall eBPF con la idea de ganar en cuanto a consumo se refiere.

Se van a dividir las pruebas en **tres escenarios**. Primero, bloqueando tráfico SIP en una situación de carga media-baja. Posteriormente, en el escenario dos y tres, se someterá al sistema a una carga alta de tráfico SIP. La idea de las distintas pruebas es, por un lado, analizar el nivel de influencia que supone para el sistema actual (*iptables*) tener que analizar largas listas de IPs (las que llamamos *whitelist*) antes de decidir sobre el tráfico en concreto que se bloquea. Esta situación se resume en la *Figura 38*:

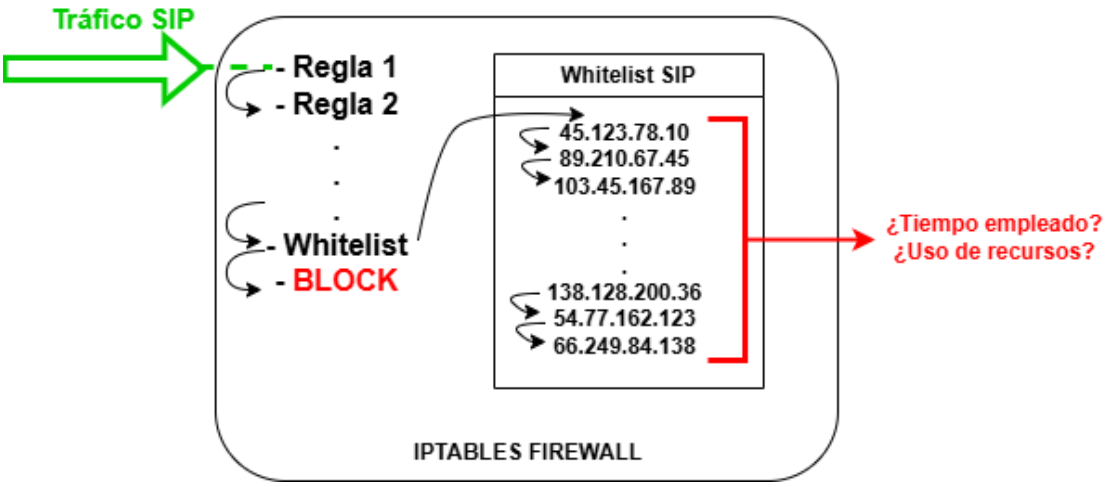


Figura 38. Situación de análisis de Whitelists por iptables

Por otro lado, y de manera más relevante, se busca analizar cómo responden los recursos del sistema, en especial el uso de la CPU, ante situaciones de alta carga de tráfico. Para evaluar el desempeño y la eficacia de eBPF en estos escenarios, se realizará una **comparación** entre el comportamiento del sistema al utilizar el **firewall basado en *iptables*** y al aplicar el **firewall desarrollado con eBPF**.

Lo mostrado en la siguiente *Figura 39* permite aclarar el proceso que toma el tráfico al alcanzar nuestro agente, pasando primero por el Firewall eBPF si está activo, y posteriormente por el tradicional *iptables*.

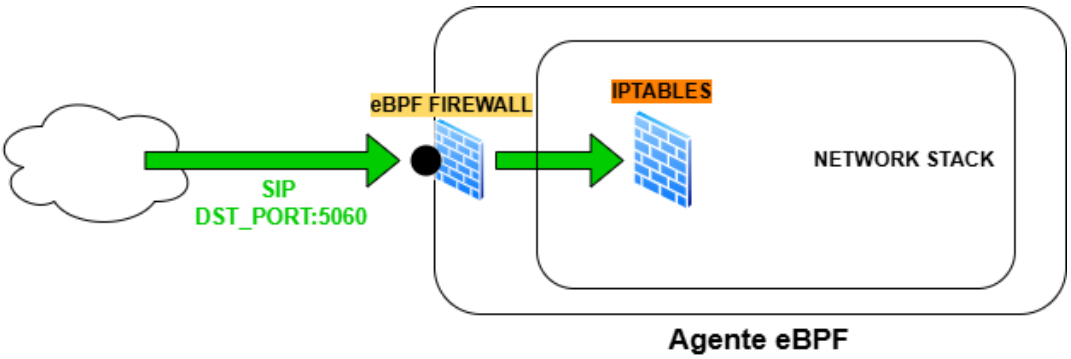


Figura 39. Proceso del tráfico a través de los dos Firewall.

Teniendo en cuenta lo citado anteriormente, se van a comparar los siguientes métodos de bloqueo de tráfico, ante mismas condiciones según el escenario:

- **Firewall iptables** bloqueando el tráfico SIP, tras analizar entre 0 y 4 *whitelists* de IPs.
- **Firewall eBPF** bloqueando el tráfico SIP.

En este entorno de pruebas **el tráfico SIP será generado por el eBPF Master** (*sipgen-ebpf ; 192.168.66.23*) **y bloqueado por el agente en el que se instala el firewall** (*firewall-ebpf ; 192.168.66.160*), utilizando el programa SIPp (funcionamiento explicado en el ANEXO I). Este escenario es el ya descrito en la *Figura 18*.

5.2.1. Escenario 1: Carga media

El primer escenario tiene el objetivo de evaluar el comportamiento del sistema bajo condiciones controladas. Se simula un escenario de tráfico moderado, y se evalúa el comportamiento de los recursos de la máquina que bloquea el tráfico SIP.

Cada prueba se ha repetido en más de tres ocasiones, con el objetivo de **obtener un promediado** que permita obtener una aproximación más realista del comportamiento del sistema.

Tráfico generado: 4.000 llamadas por segundo **Tiempo de ejecución: ~ 120 segundos** **Llamadas totales generadas: ~ 480.000 llamadas** **192.168.66.23 ----> 192.168.66.160 (5060)**

Tabla de resultados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists)	2.78 GB	1.05%	8.63%	**7.58%**
IPTABLES (2 Whitelists)	2.78 GB	1.26%	8.16%	**6.90%**
Firewall eBPF	2.79 GB	1.26%	5.82%	**4.56%**

Se observa que, al aplicar el Firewall eBPF, se está implicando a **menos recursos** en cuanto a **CPU** se refiere.

Ganancia relativa de CPU, respecto a la mejor situación analizada con *iptables*:

6.90% - 4.56% = 2.34% → Implica **reducir en un ~34% el uso de CPU**.

En la siguiente *Figura 40* se observa de forma gráfica el **uso de CPU que alcanzó la máquina en este escenario de tráfico** y ante las situaciones analizadas (gráficas obtenidas de Proxmox):

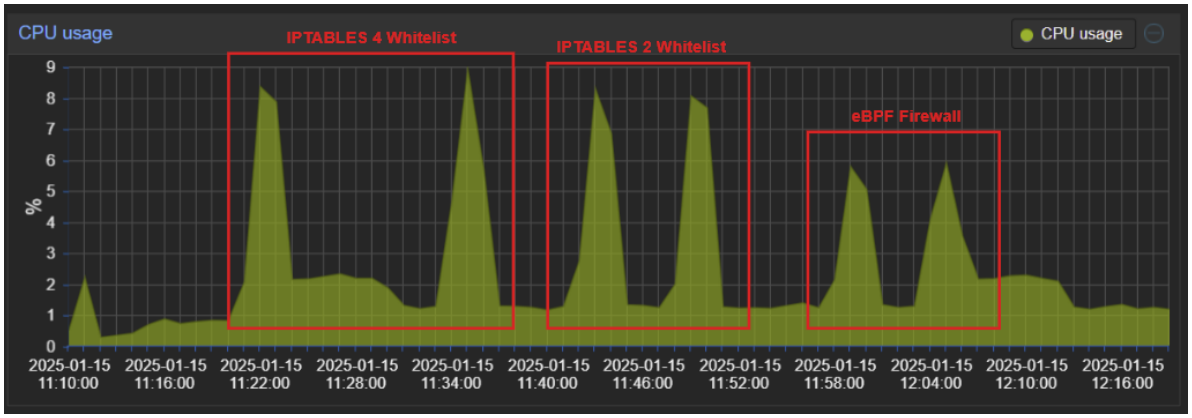


Figura 40. Uso de CPU ante las distintas situaciones.

En la *Figura 41* se muestran las estadísticas presentadas por SIPp tras las pruebas de este escenario. Caben destacar (remarcados) los valores de tiempo total en ejecución de la herramienta, frecuencia de llamadas por segundo, y total de llamadas generadas. Por otro lado, se observa en el flujo de mensajes SIP superior, como los INVITE no reciben ningún tipo de respuesta al estar rechazándose el tráfico.

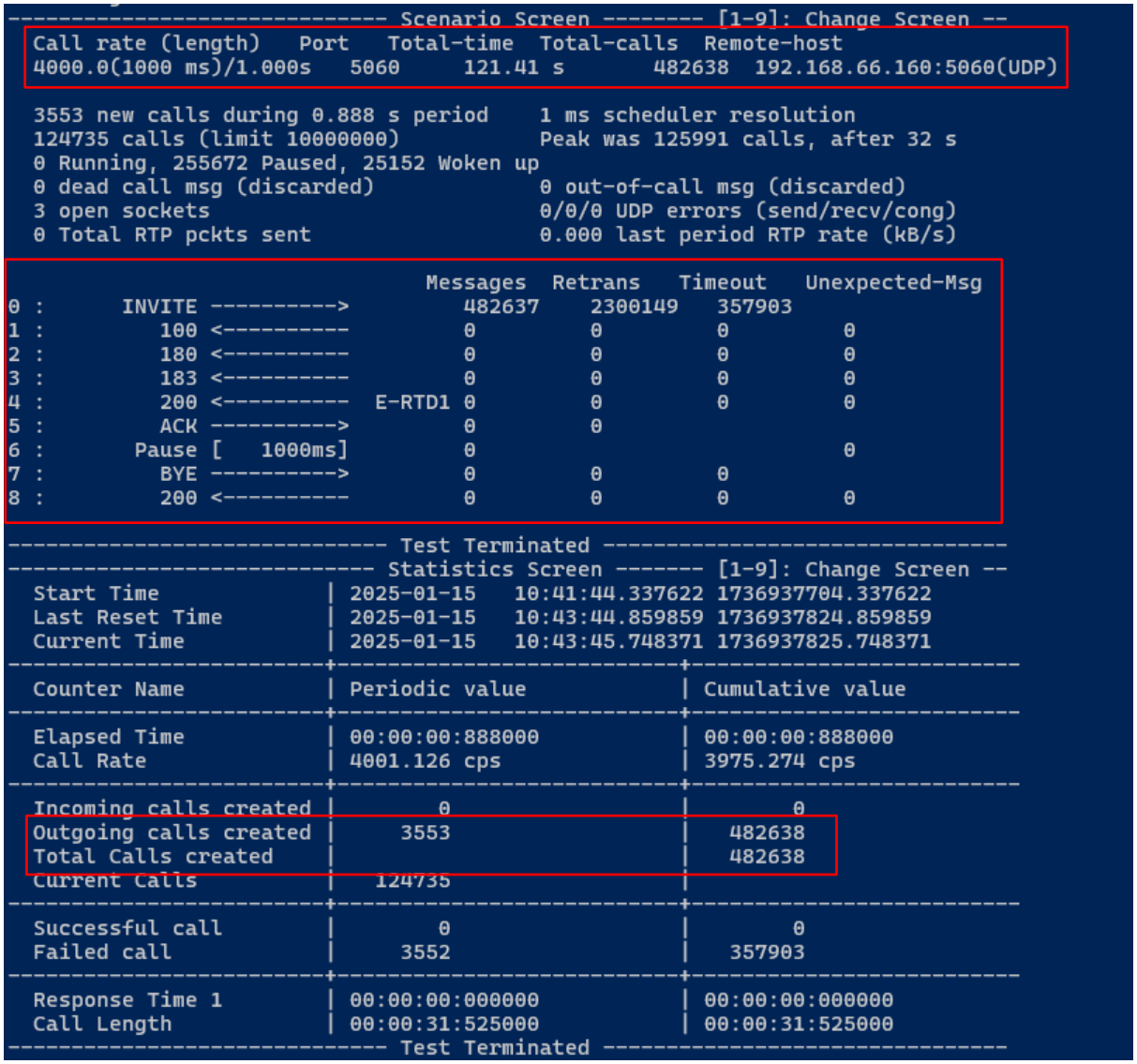


Figura 41. Estadísticas volcadas por SIPp.

Además, se puede visualizar el comportamiento de *iptables* para las distintas situaciones, visualizando en directo los paquetes que hacen "match" con cada regla, utilizando el comando: *watch -n 1 "sudo iptables -L -n -v"*
En este ejemplo (*Figura 42*), se comprueba cómo los paquetes están siendo bloqueados por la regla (en rojo) que se evalúa después de analizar las cuatro *whitelists* (en amarillo). A la izquierda, también marcado en rojo, aparece el número de paquetes bloqueados. El ejemplo mostrado coincide con la primera situación de bloqueo analizada.

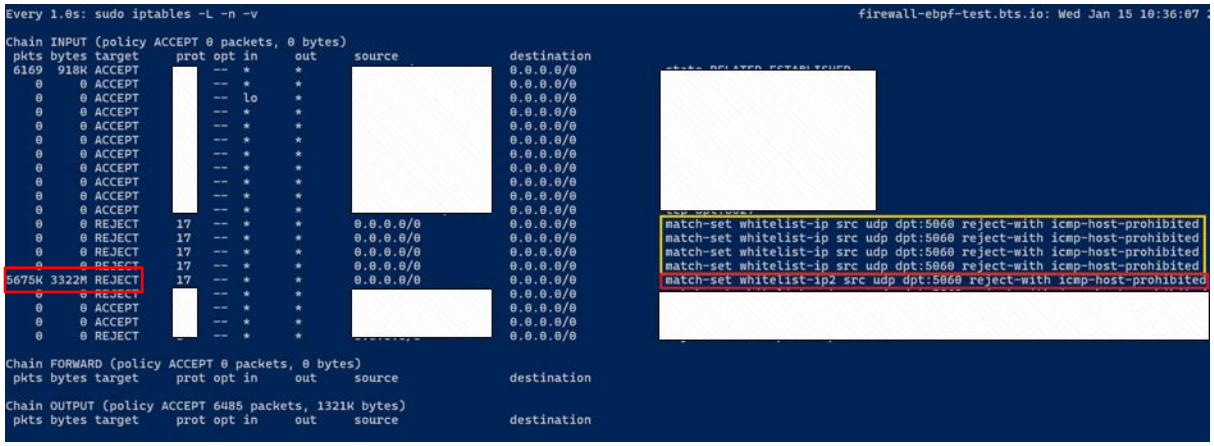


Figura 42. iptables y estado de las whitelists.

Se pueden consultar las mismas figuras para otros escenarios en el ANEXO H. **Las conclusiones se desarrollarán tras las pruebas de los tres escenarios, en el próximo capítulo.**

5.2.2. Escenario 2: Carga alta no prolongada

En esta segunda fase de pruebas, se escaló la carga de trabajo con el objetivo de evaluar el rendimiento del sistema bajo condiciones de alto estrés. Simulando un escenario de 40.000 llamadas por segundo (diez veces el volumen de las pruebas iniciales) durante el mismo periodo de tiempo, se buscó determinar la capacidad del sistema en las distintas situaciones para gestionar una demanda significativamente mayor.

Se replicaron las configuraciones de las pruebas anteriores (*iptables* con dos y cuatro *whitelists*, y eBPF) para establecer comparativas precisas.

Tráfico generado: 40.000 llamadas por segundo
Tiempo de ejecución: ~ 120 segundos
Llamadas totales generadas: ~ 1.3M llamadas
192.168.66.23 ----> 192.168.66.160 (5060)

Tabla de resultados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists)	2.83 GB	1.67%	17.21%	**15.54%**
IPTABLES (2 Whitelists)	2.82 GB	1.29%	15.70%	**14.41%**
Firewall eBPF	2.82 GB	1.26%	9.81%	**8.55%**

La subida de CPU tras someterse al tráfico SIP es significativamente mayor en todas las situaciones que en el escenario anterior, ya que la carga también lo es. En este caso se comprueba que **aplicar eBPF en situaciones de alta carga también implica mejoras en cuanto a CPU** se refiere.

Ganancia relativa de CPU, respecto a la mejor situación analizada con *iptables*:

14.41% - 8.55% = 5.86% → Implica **reducir en un ~40% el uso de CPU**.

De igual forma que para el escenario anterior, en la siguiente *Figura 43* se observa de forma gráfica el **uso de CPU que alcanzó la máquina en este escenario de tráfico**:

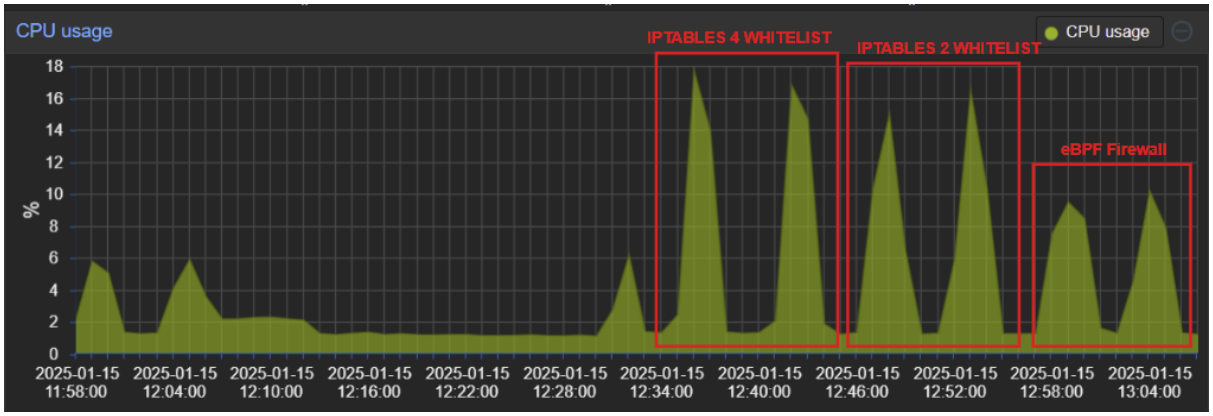


Figura 43. Uso de CPU ante las distintas situaciones.

De nuevo, se presentan las estadísticas que nos devuelve **SIPp** ante estas pruebas (*Figura 44*), alcanzando el 1.2M de llamadas totales generadas:

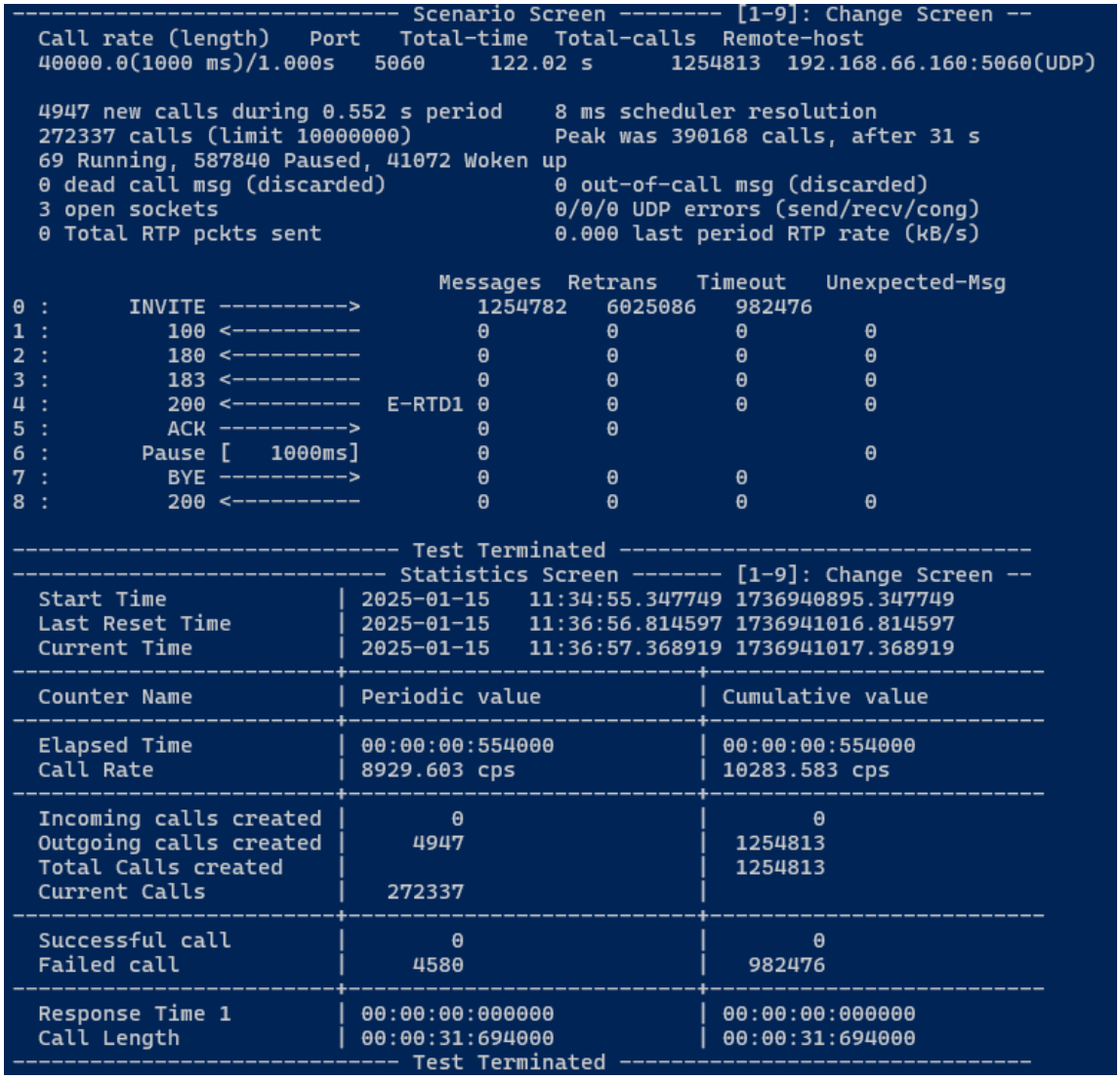


Figura 44. Estadísticas volcadas por SIPp.

La cantidad de tráfico bloqueado para cada situación y el estado de *iptables* se puede consultar en el ANEXO H.

5.2.3. Escenario 3: Carga alta prolongada

En este último escenario se quiere someter al sistema a una situación de **carga alta prolongada durante 5 minutos**, para comparar el desempeño de los distintos sistemas de firewall ante una situación extrema. De nuevo, se compara el uso de *iptables* con el de eBPF, pero para estas pruebas finales se incluirá una nueva situación: firewall *iptables* bloqueando el tráfico directamente, **sin analizar ninguna *Whitelist***.

De esta forma, aunque no es el funcionamiento habitual de los equipos de BTS, llevamos la situación a su **punto más crítico de comparación**, ya que el firewall convencional con *iptables* **mostrará su desempeño máximo** al no tener que "gastar" recursos en analizar las largas listas de IPs, con lo que se ha demostrado que ello implica. Así, se podrá comparar la "mejor situación" para *iptables*, con el uso de eBPF.

Tráfico generado: 40.000 llamadas por segundo
Tiempo de ejecución: ~ 300 segundos
Llamadas totales generadas: ~ 2.5M llamadas
192.168.66.23 ----> 192.168.66.160 (5060)

Tabla de resultados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists)	2.83 GB	1.27%	16.97%	**15.70%**
IPTABLES (2 Whitelists)	2.82 GB	1.19%	15.44%	**14.25%**
IPTABLES (Bloqueo directo)	2.83 GB	1.25%	14.08%	**12.83%**
Firewall eBPF	2.82 GB	1.18%	9.76%	**8.58%**

De nuevo, **el Firewall eBPF vuelve a mejorar las prestaciones** del sistema convencional con *iptables*, incluso en situaciones de carga prolongada como esta. Incluso en la comparación con la mejor situación con *iptables*, eBPF introduce un **ahorro significativo** de recursos en cuanto a gasto de CPU.

Aun así, también se comprueba en este escenario que la tarea de **análisis de las *Whitelist* SIP** con gran cantidad de IPs, hace **aumentar el gasto de recursos** a la máquina que bloquea el tráfico. Comparando el bloqueo directo con la primera situación, el sistema invierte **casi un 3% menos de CPU en el procesado de paquetes** (lo que supone **reducirlo en casi un 20%**).

Ganancia relativa de CPU, respecto a la mejor situación analizada con *iptables*:

12.83% - 8.58% = 4.25% → Implica **reducir en un ~33% el uso de CPU**.

De igual forma que para el escenario anterior, en la siguiente *Figura 45* podemos observar de **forma gráfica el uso de CPU que alcanzó la máquina en este escenario de tráfico**:

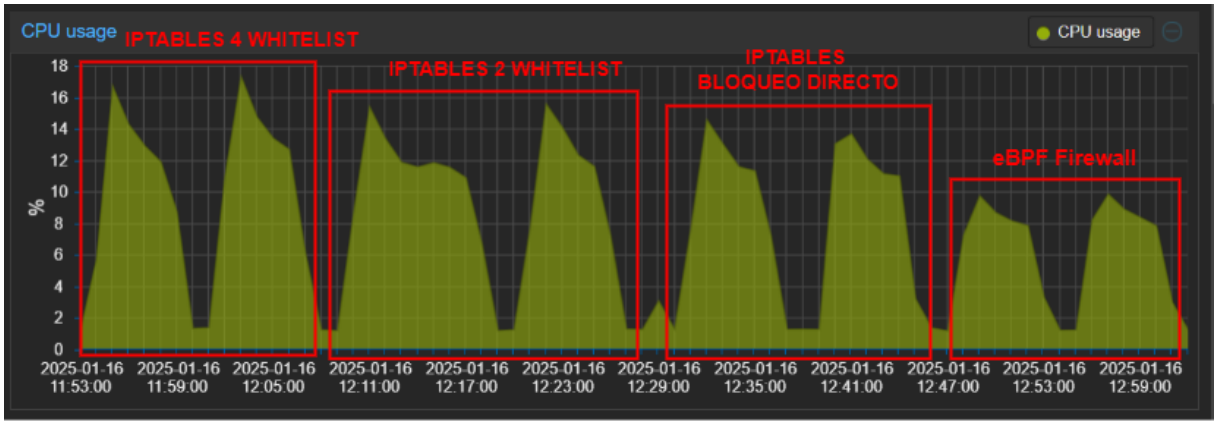


Figura 45. Uso de CPU ante las distintas situaciones.

A continuación la *Figura* 46 muestra las estadísticas de SIPp tras los 5 minutos de llamadas simuladas, llegando a generar más de 2.5 millones:

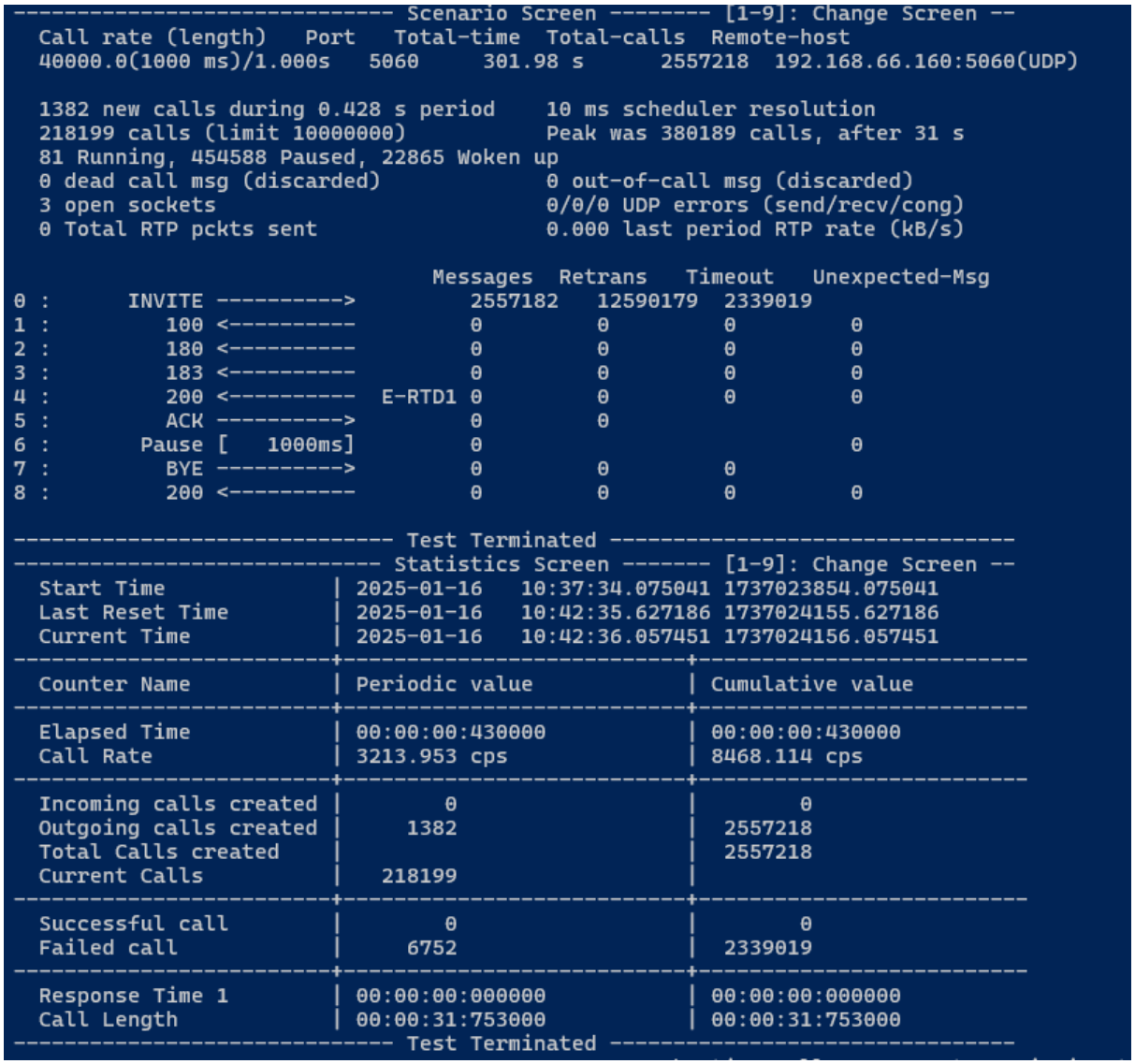


Figura 46. Estadísticas volcadas por SIPp.

La cantidad de tráfico bloqueado para cada situación y el estado de *iptables* se puede consultar en el ANEXO H.

Capítulo 6: Conclusiones y líneas futuras

6.1. Conclusiones

El objetivo general de este proyecto ha sido el estudio y análisis de eBPF como tecnología, junto con su aplicación práctica mediante el uso de lenguajes de programación como C y Python. Esto ha permitido desarrollar una solución personalizada, como el Firewall eBPF, y explorar la integración de la tecnología en entornos reales.

El desarrollo e implementación del Firewall eBPF ha permitido evidenciar una serie de ventajas y beneficios clave que hacen de esta solución una propuesta altamente eficiente y práctica en entornos reales. A lo largo del proyecto y las pruebas de carga realizadas, se han obtenido resultados positivos, como el ahorro de más del 30% de uso de CPU en situaciones de alta carga de tráfico, lo cual subraya el valor añadido de esta tecnología.

Una de las principales ventajas del Firewall eBPF es su facilidad de **configuración**, ya que no requiere realizar ajustes complejos en equipos de red importantes o en el propio Linux. Esta simplicidad es complementada por una **gestión centralizada e intuitiva** mediante una web, que permite a trabajadores con menos experiencia en la administración de firewalls operar de manera eficiente, sin comprometer la seguridad ni la funcionalidad del sistema.

Además, su **escalabilidad** lo convierte en una solución ideal para entornos con múltiples máquinas, ya que basta con configurar los agentes eBPF para extender su alcance. Esto permite aplicar políticas de seguridad de manera uniforme y eficaz en toda la infraestructura. Por otro lado, el firewall mantiene la **compatibilidad con las funcionalidades tradicionales de *iptables***, incluyendo la posibilidad de gestionar *whitelists* a través de bases de datos, asegurando una integración fluida con sistemas ya existentes.

Al realizar la comparación con el sistema actual y centrarse en términos de **recursos**, el ahorro de CPU observado es significativo, ya que libera capacidad en las máquinas, lo que incrementa la escalabilidad y optimiza el rendimiento general de los sistemas. Además, el Firewall eBPF ha demostrado ser una herramienta sólida frente a **situaciones de alta carga de tráfico**, como puede ser un incremento importante de tráfico proveniente de un cliente/proveedor o los ataques DDoS, donde su capacidad para filtrar y bloquear paquetes de manera eficiente protege la infraestructura crítica de BTS [14].

Finalmente, los resultados obtenidos abren la puerta a futuras aplicaciones de eBPF en otros puntos de la infraestructura. Su flexibilidad y rendimiento sugieren que puede ser la base de nuevos proyectos orientados a mejorar tanto el monitoreo como la seguridad en sistemas complejos, maximizando el potencial de esta tecnología.

6.2. Líneas futuras

A partir del desarrollo y las pruebas realizadas en este proyecto, se abren numerosas oportunidades para **extender las funcionalidades** del Firewall eBPF y explorar **nuevos usos de esta tecnología** en la infraestructura de red. Algunas de las líneas futuras más relevantes son:

1. Funcionalidades adicionales al firewall
Se podrían incorporar herramientas para un análisis más profundo y detallado del tráfico gestionado por el firewall. Por ejemplo:
- Implementar un **contador de paquetes procesados** por cada regla, lo que permitiría entender mejor qué reglas son más utilizadas y optimizar su configuración.
- Extraer **métricas** derivadas de este análisis para supervisar patrones de tráfico y detectar posibles anomalías o ineficiencias.
- Añadir **soporte para netmasks** en las reglas del firewall, habilitando la capacidad de bloquear o permitir tráfico de redes completas en lugar de direcciones IP individuales, aumentando la versatilidad del sistema.

2. Limitador de tráfico con eBPF
Una funcionalidad avanzada que sería interesante implementar es un sistema limitador de tráfico (*rate limit*) basado en eBPF. Esto permitiría:
- **Restringir el número de llamadas por segundo** desde ciertas fuentes, protegiendo los sistemas internos frente a cargas excesivas.
- **Mitigar ataques DDoS** limitando el tráfico malicioso antes de que alcance recursos críticos.
Las pruebas realizadas han demostrado que eBPF es altamente eficiente para bloquear tráfico, lo que lo convierte en una herramienta ideal para este tipo de protección.

3. Redirección de tráfico entre interfaces
Se podría aprovechar la funcionalidad de **XDP_REDIRECT** para redirigir el tráfico entre distintas interfaces de un equipo. Esto permitiría:
- Crear soluciones para análisis de tráfico "offline", **desviando copias del tráfico hacia sistemas dedicados al monitoreo y análisis** sin afectar al rendimiento del tráfico principal.
- Implementar **funciones similares a los TAPs** (*Test Access Points*), que replican el tráfico para inspección o análisis sin interferir con las operaciones diarias de la red.

Finalmente, una vez se completase la configuración con lo que requiere y aplicando las funcionalidades mencionadas, el objetivo próximo más claro sería el de integrar la solución del Firewall eBPF a los equipos en producción de la empresa.

Estas líneas futuras no solo expanden las capacidades del Firewall eBPF, sino que también plantean nuevos usos para eBPF en la optimización y protección de sistemas de red.

Capítulo 7: Anexos

ANEXO A. Protocolo SIP.

SIP [5] es un protocolo basado en texto que facilita el establecimiento, modificación y finalización de sesiones de comunicación multimedia entre dos o más participantes [5]. Estas comunicaciones pueden ser llamadas de voz, videoconferencias, mensajería instantánea o transferencia de archivos. Se utiliza, mayoritariamente, en el mundo de la telefonía IP. Cabe destacar cómo su mecanismo de petición-respuesta facilita la resolución de errores.

Los mensajes SIP describen la identidad de los participantes en una llamada y cómo los participantes pueden ser localizados sobre una red IP. Una vez que el intercambio de los mensajes de configuración es completado, la comunicación multimedia utiliza otro protocolo, típicamente **RTP** (Real-Time Transmission Protocol).

El protocolo utiliza mensajes para establecer, modificar y finalizar sesiones de comunicación en redes IP. Entre los principales mensajes destacan:

INVITE: Inicia una sesión solicitando una conexión con un usuario o recurso.
ACK: Confirma la recepción de una respuesta exitosa al INVITE.
BYE: Finaliza una sesión activa entre dos usuarios.
CANCEL: Cancela una solicitud que aún no ha sido respondida.
REGISTER: Registra la ubicación de un usuario en un servidor SIP, permitiendo su localización.
OPTIONS: Consulta las capacidades del otro extremo sin iniciar una sesión.

Estos mensajes van acompañados de respuestas numéricas, similares a los códigos HTTP, que indican el estado de las solicitudes (por ejemplo, 200 OK para éxito, 404 Not Found para usuario no disponible). Este sistema asegura una comunicación clara y eficiente en redes VoIP.

ANEXO B. Proyectos, patentes y empresas que utilizan eBPF.

En este anexo se nombran y describen algunas soluciones existentes hoy en día en la industria tecnológica que utilizan eBPF para su funcionamiento. Muchos de ellos, y más, pueden consultarse en la web oficial de eBPF:
https://ebpf.io/applications/

- **Proyectos destacados basados en eBPF**:

Falco
Es un monitor de actividad de comportamiento diseñado para detectar actividad anómala en aplicaciones. Falco **audita un sistema en la capa del *kernel* de Linux** con la ayuda de eBPF. Enriquece los datos recopilados con otros flujos de entrada, como métricas de tiempo de ejecución del contenedor y métricas de Kubernetes, y permite supervisar y detectar continuamente la actividad del contenedor, la aplicación, el host y la red.
https://falco.org/tags/ebpf/

Katran
Katran es una biblioteca C++ y un programa eBPF para construir un plano de reenvío de **equilibrio de carga de capa 4** de alto rendimiento. Katran aprovecha la infraestructura **XDP** del *kernel* de Linux para proporcionar una instalación en el *kernel* para el procesamiento rápido de paquetes. Su rendimiento se escala linealmente con el número de colas de recepción de la NIC y utiliza encapsulación RSS amigable para el reenvío a equilibradores de carga de capa 7.
https://github.com/facebookincubator/katran

Cilium
Cilium es un proyecto de código abierto que proporciona **redes, seguridad y observabilidad** impulsadas por eBPF. Se ha diseñado específicamente desde cero para llevar las **ventajas de eBPF al mundo de Kubernetes** y abordar los nuevos requisitos de escalabilidad, seguridad y visibilidad de las cargas de trabajo de los contenedores.
https://cilium.io/

Tracee
Tracee utiliza la tecnología eBPF para detectar y filtrar eventos del sistema operativo, ayudándole a **exponer perspectivas de seguridad, detectar comportamientos sospechosos y capturar indicadores forenses**.
https://github.com/aquasecurity/tracee

- **Patentes que aplican eBPF**:

Load balancing IPsec tunnel processing with extended Berkeley packet filter (eBPF)
[US10623372B2]
https://patents.google.com/patent/US10623372B2/en?q=(ebpf)&oq=ebpf

Anomaly detection for microservices
[US11811801B2]
https://patents.google.com/patent/US11811801B2/en?q=(ebpf)&oq=ebpf&page=1

File protection method and device based on eBPF, equipment and medium
[CN116415300A]
https://worldwide.espacenet.com/patent/search/family/087055994/publication/CN116415300A?q=pn%3DCN116415300A

- **Empresas punteras que utilizan eBPF**

Google
Google uses eBPF for security auditing, packet processing, and performance monitoring.
https://netdevconf.info//0x14/session.html?talk-replacing-HTB-with-EDT-and-BPF

Netflix
Netflix uses eBPF at scale for network insights [15]
https://netflixtechblog.com/how-netflix-uses-ebpf-flow-logs-at-scale-for-network-insight-e3ea997dca96

Android
Android uses eBPF to monitor network usage, power, and memory profiling.
https://source.android.com/docs/core/architecture/kernel/bpf?hl=es

Shopify
Shopify uses eBPF through Falco for intrusion detection.
https://www.youtube.com/watch?v=6pVci31Mb6Q

ANEXO C. PostgreSQL - Gestión y comandos.

Secuencia de comandos para consultar la *Whitelist* SIP

```
cirigoyen@firewall-ebpf-test:~$ sudo -i -u postgres
postgres@firewall-ebpf-test.bts.io:~$ psql
psql (15.10 (Debian 15.10-0+deb12u1))
postgres=# \c sip_whitelist
You are now connected to database "sip_whitelist" as user "postgres".
sip_whitelist=# select * from direcciones_ip ;
   ip_address
-----------------
 192.168.67.7
 100.100.100.100
 199.20.20.20
(3 rows)
```

Acceso a la base de datos desde los programas

```
# Configuración de la base de datos
DB_CONFIG = {
    'dbname': 'firewall_users',
    'user':'xxxxxxx',
    'password':,'xxxxxx'
    'host': 'localhost',
    'port': '5432'
}
# Función para obtener la conexión a la base de datos
def get_db_connection():
    return psycopg2.connect(**DB_CONFIG)
```

Consulta de la tabla con usuarios autorizados

Tal como se puede observar, las contraseñas no se almacenan en texto plano, sino que se guardan de forma codificada para garantizar una mayor seguridad.

```
firewall_users=# select * from users;
 id |  username  |                          password
----+------------+------------------------------------------------------------
  4 | jlazaro    | $2b$12$1.7rKWBwqEMQiukTUQRmmOKfY12nevOOEOxs2Jpd6/k32g1UqNdmu
  5 | cirigoyen  | $2b$12$QrPmYZe0VhgH6f1EtELioekidtVLDg0vnkW4B9vJkjzoiRMUTt9Ku
  6 | jmoranchel | $2b$12$jMMIGoHUCeAaubPwEFmfWOFtn2TpGR9tG7CACJZAC7drGbmiFhNwy
(3 rows)
```

Figura 47. Tabla en PostgreSQL de usuarios autorizados.

ANEXO D. Proceso de instalación de herramientas utilizadas

Este anexo está dedicado a profundizar en el proceso de instalación y configuración de los equipos del entorno de pruebas, así como de las principales herramientas de las que se ha hecho uso en el desarrollo del proyecto, siguiendo con lo comentado en el *Capítulo 3*.

El primer paso es la creación de las máquinas virtuales que se utilizarán para el proyecto (*firewall-ebpf-test* y *sipgen-ebpf-test*) en los servidores que las alojarán. Estos servidores funcionan con Proxmox, y la tarea de crear las máquinas virtuales es llevada a cabo por el equipo de sistemas de BTS. Este proceso sigue una serie de pasos organizados y específicos.

Es fundamental identificar los servidores capaces de alojar estas máquinas, teniendo en cuenta factores como los recursos disponibles y las capacidades de conectividad. Tras este análisis, se decide alojar las máquinas en dos servidores de Hetzner, denominados "pve1-hetz" y "*pve4-hetz*". La *Figura 48* muestra la composición del Data-Center en cuestión, que consta de un total de cuatro servidores Proxmox.

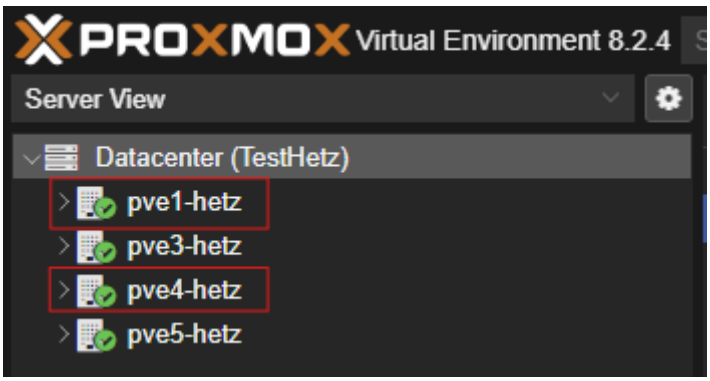


Figura 48. Nodos de Proxmox.

Seleccionados los servidores que actuarán como "host", se procede a la creación de las máquinas virtuales.
Estas han sido creadas con una plantilla predefinida, y aplicando la imagen de **Debian 12**.

Con las máquinas virtuales preparadas, el primer paso en su configuración es habilitar la conectividad de red. Ambas máquinas se configuran con direcciones IP privadas dentro de la misma red: 192.168.66.0/24, la red privada compartida entre los servidores de Hetzner. Esto asegura que puedan comunicarse entre sí y realizar las pruebas de tráfico SIP sin inconvenientes.

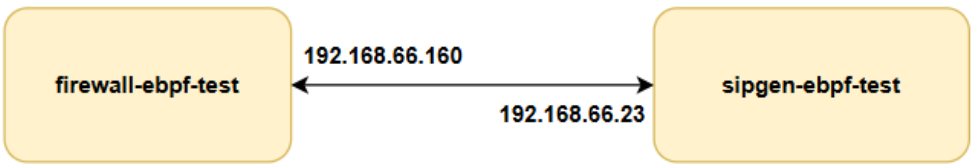


Figura 49. Comunicación entre máquinas.

En la siguiente captura, extraída de Proxmox, se puede consultar la configuración de Hardware que se ha aplicado a ambas máquinas de pruebas.

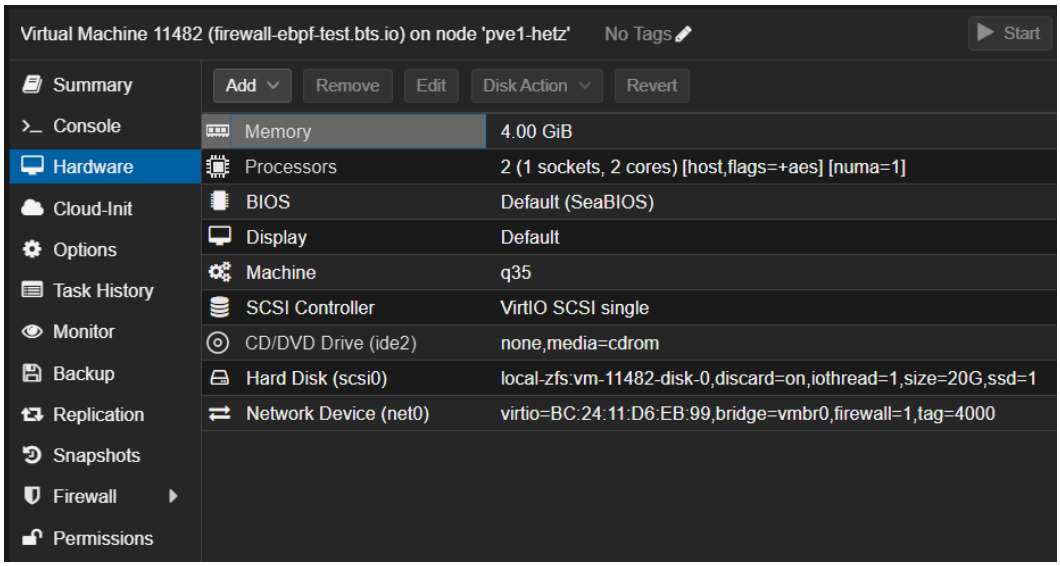


Figura 50. Hardware configurado en las máquinas.

A partir de aquí, se detalla el proceso de instalación necesario para las herramientas de Software necesarias:

eBPF

- *clang* y *llvm*: Compiladores para convertir código C en bytecode eBPF. Los programas eBPF suelen escribirse en C
- *elfutils-libelf-devel*: Los programas eBPF son cargados en el *kernel* como ELF. Esta librería permite inspeccionar y manejar estos archivos para depuración y compilación.
- *gcc* y *gcc-c++*: Compiladores para C y C++. Similar a *clang*
- *bpftool*: Herramienta para interactuar con programas eBPF cargados en el *kernel*
- *kernel-devel* y *kernel-headers:* Headers y fuentes del *kernel* necesarios para desarrollar y cargar programas eBPF
- *libbpf* y *libbpf-devel*: Librería para interactuar con programas eBPF desde espacio de usuario

Estos paquetes han sido instalados desde la consola de comandos de la máquina, mediante:

 # sudo apt install clang llvm elfutils-libelf-devel gcc gcc-c++ bpftool kernel-devel libbpf libbpf-devel

BCC

- *bpfcc-tools*: Este paquete incluye un conjunto de herramientas predefinidas que utilizan BCC para interactuar con programas eBPF
- *libbpfcc-dev*: Contiene las cabeceras de desarrollo y las bibliotecas necesarias para trabajar con BCC desde lenguajes como Python o C

Estos paquetes, de igual forma que para eBPF, se instalan en Debian 12 mediante:

 # sudo apt install bpfcc-tools libbpfcc-dev

XDP

- *iproute*: Herramienta que incluye el comando *ip*, necesario para adjuntar programas XDP a interfaces de red

El comando para instalarla es el siguiente:

 # sudo apt install iproute

Python

- *python3* y *python3-pip*: Instala el intérprete de Python y *pip* para instalar bibliotecas adicionales
- *libbcc, libbcc-dev, python3-bcc*: Biblioteca y conjunto de herramientas de alto nivel para trabajar con eBPF desde Python.
- *pip install ebpf*: Instalar directamente en Python una biblioteca ligera y moderna para trabajar con eBPF, incluyendo XDP, TC y otras funciones.
- *python3-jsonschema:* Para utilizar JSON Schemas

Comandos de instalación:

sudo apt install python3 python3-pip libbcc libbcc-dev python3-bcc python3-jsonschema

Flask

- *flask:*

 # pip install flask

PostgreSQL, SSL

- *postgresql* y *postgresql-contrib*: Sistema de gestión de bases de datos PostgreSQL
- *python3-psycopg2* y *python3-bcrypt*: Adaptador Psycopg 2 para interactuar con bases de datos PostgreSQL desde Python y biblioteca de hashing de contraseñas.
- *libssl-dev*: Librería de desarrollo para OpenSSL, utilizada para implementar protocolos de seguridad y cifrado como TLS/SSL.

 # sudo apt install postgresql postgresql-contrib
 # sudo apt install libssl-dev
 # sudo apt install python3-psycopg2 python3-bcrypt

SIPp

 # git clone https://github.com/SIPp/sipp
 # cd sipp/
 # cmake . -DUSE_SSL=1 -DUSE_SCTP=1 -DUSE_PCAP=1
 # make
 # export PATH=$PATH:/usr/local/bin
 # sudo make install

Comprobar la correcta instalación de SIPp:

 # sipp -v

ANEXO E. Desarrollo del código.

El código completo con todas sus partes queda subido a un repositorio público de *Github* para su consulta: https://github.com/cirigoyen277/firewall_ebpf_cirigoyen

Sección 1. Programa eBPF para el análisis de paquetes

```
#include <linux/bpf.h>
#include <linux/if_ether.h>
#include <linux/ip.h>
#include <linux/tcp.h>
#include <linux/udp.h>
```

```
struct rule_key_proto {
   u8 proto;
};
struct rule_key_ipsrc {
   u32 ip_src;
};
struct rule_key_ipsrc_proto_portdst {
   u8 proto;
   u32 ip_src;
   u16 port_dst;
};
```

```
BPF_HASH(allowed_rules1, struct rule_key_proto, u32);
BPF_HASH(allowed_rules2, struct rule_key_ipsrc, u32);
BPF_HASH(allowed_rules3, struct rule_key_ipsrc_proto_portdst, u32);
BPF_HASH(blocked_rules1, struct rule_key_proto, u32);
BPF_HASH(blocked_rules2, struct rule_key_ipsrc, u32);
BPF_HASH(blocked_rules3, struct rule_key_ipsrc_proto_portdst, u32);
```

```
 //////// EVITAR ACCESOS INVALIDOS EN MEMORIA //////////
   // Obtenemos los límites de los datos del paquete para evitar accesos inválidos.
   void *data = (void *)(long)ctx->data; // Inicio del paquete
   void *data_end = (void *)(long)ctx->data_end; // Fin del paquete
   // Verificar si el paquete tiene al menos la cabecera Ethernet completa
   struct ethhdr *eth = data;
   if ((void*)(eth + 1) > data_end)
      return XDP_PASS; // Si no, pasamos el paquete (no se procesa más)
   // Verificar si el paquete tiene un protocolo IP
   if (eth->h_proto != htons(ETH_P_IP))
      return XDP_PASS; // Si no es IP, pasamos el paquete
   // Verificar si el paquete tiene al menos la cabecera IP completa
```

```
   struct iphdr *ip = data + sizeof(struct ethhdr); // Cabecera IP comienza después de Ethernet
   if ((void*)(ip + 1) > data_end)
      return XDP_PASS; // Si no, pasamos el paquete
   // Verificar si el protocolo es TCP o UDP
   if (ip->protocol != IPPROTO_TCP && ip->protocol != IPPROTO_UDP && ip->protocol !=
IPPROTO_ICMP)
      return XDP_PASS; // Si no es TCP o UDP, pasamos el paquete
   // Verificar si el paquete tiene al menos la cabecera TCP/UDP completa
   struct tcphdr *tcp = (void *)ip + sizeof(struct iphdr); // Cabecera TCP/UDP después de IP
   if ((void*)(tcp + 1) > data_end)
      return XDP_PASS; // Si no, pasamos el paquete
```

```
   struct rule_key_proto key1 = {};
   key1.proto = ip->protocol;

   struct rule_key_ipsrc key2 = {};
   key2.ip_src = ip->saddr;

   struct rule_key_ipsrc_proto_portdst key3 = {};
   key3.proto = ip->protocol;
   key3.ip_src = ip->saddr;
   key3.port_dst = tcp->dest;
```

```
   //////// CHECK WHITELIST ////////
   u32 *allow_value1 = allowed_rules1.lookup(&key1);
   if (allow_value1) {
      return XDP_PASS; // Si está en la whitelist, permite el tráfico
   }

   u32 *allow_value2 = allowed_rules2.lookup(&key2);
   if (allow_value2) {
      return XDP_PASS; // Si está en la whitelist, permite el tráfico
   }

   u32 *allow_value3 = allowed_rules3.lookup(&key3);
   if (allow_value3) {
      return XDP_PASS; // Si está en la whitelist, permite el tráfico
   }

   //////// CHECK BLACKLIST ////////
   u32 *block_value1 = blocked_rules1.lookup(&key1);
   if (block_value1) {
      return XDP_DROP; // Si está en la darklist, bloquea el tráfico
   }
```

```
    u32 *block_value2 = blocked_rules2.lookup(&key2);
    if (block_value2) {
        return XDP_DROP; // Si está en la darklist, bloquea el tráfico
    }

    u32 *block_value3 = blocked_rules3.lookup(&key3);
    if (block_value3) {
        return XDP_DROP; // Si está en la darklist, bloquea el tráfico
    }

    // Por defecto, se bloquea el paso
    return XDP_DROP;
```

Sección 2. Programa backend para los agentes: “add_rule.py”

```
from flask import Flask, request, jsonify
from bcc import BPF
import socket
import struct
import ctypes
import signal
import sys
import os
import psycopg2

app = Flask(__name__)

INTERFACE = "enp6s18"
```

```
b = BPF(text=bpf_program)
fn = b.load_func("block_packet", BPF.XDP)
b.attach_xdp(INTERFACE, fn)
```

```
# Mapas de reglas bloqueadas
blocked_rules1 = b.get_table("blocked_rules1")
blocked_rules2 = b.get_table("blocked_rules2")
blocked_rules3 = b.get_table("blocked_rules3")

# Mapas de reglas permitidas
allowed_rules1 = b.get_table("allowed_rules1")
allowed_rules2 = b.get_table("allowed_rules2")
allowed_rules3 = b.get_table("allowed_rules3")
```

```
# Función para limpiar el programa eBPF al salir
```

```
def cleanup(sig, frame):
    print("\nDesasociando el programa eBPF de la interfaz...")
    b.remove_xdp(INTERFACE, 0)
    print("Programa desasociado. Saliendo.")
    sys.exit(0)

# Registrar la función cleanup para las señales SIGINT y SIGTERM
signal.signal(signal.SIGINT, cleanup)
signal.signal(signal.SIGTERM, cleanup)
```

```
def ip_to_u32_inverted(ip_str):
    ip_bytes = socket.inet_aton(ip_str)  # Convierte IP a bytes
    ip_inverted = ip_bytes[::-1]  # Invierte los bytes
    return struct.unpack("!I", ip_inverted)[0]  # Convierte los bytes invertidos a u32
```

```
whitelist_all_file = "whitelist_all.txt"
if os.path.exists(whitelist_all_file):
    with open(whitelist_all_file, "r") as file:
        for line in file:
            ip = line.strip()  # Elimina espacios y saltos de línea
            if ip:  # Si la línea no está vacía
                ip_src_u32 = ip_to_u32_inverted(ip)
                key = ctypes.c_uint32(ip_src_u32)
                allowed_rules2[key] = ctypes.c_uint32(1)
else:
    print(f"El archivo '{whitelist_all_file}' no existe. No se cargaron reglas adicionales.")
```

```
def generar_whitelist_sip():
    conexion = psycopg2.connect(
        host="localhost",       # Ejemplo: "localhost"
        database="sip_whitelist",     # Nombre de la base de datos
        user="xxxxxxx",    # Usuario de la base de datos
        password= "xxxxxxx" # Contraseña del usuario
    )
    # Consulta para obtener las direcciones IP
    consulta = "SELECT host(ip_address) FROM direcciones_ip;"
    # Ruta del archivo de texto
    ruta_archivo = "whitelist_sip.txt"
    try:
        # Crear un cursor para ejecutar la consulta
        with conexion.cursor() as cursor:
            cursor.execute(consulta)
            direcciones = cursor.fetchall()  # Obtener todas las filas
```

```
        # Abrir el archivo en modo de escritura
        with open(ruta_archivo, 'w') as archivo:
            for direccion in direcciones:
                archivo.write(direccion[0] + "\n")  # Escribir cada IP en una línea
        print(f"Direcciones IP exportadas a {ruta_archivo} exitosamente.")
    except Exception as e:
        print("Ocurrió un error:", e)
    finally:
        conexion.close()  # Cerrar la conexión a la base de datos
```

```
whitelist_sip_file = "whitelist_sip.txt"
if os.path.exists(whitelist_sip_file):
    with open(whitelist_sip_file, "r") as file:
        for line in file:
            ip = line.strip()  # Elimina espacios y saltos de línea
            if ip:  # Si la línea no está vacía
                default_allowed_rules.append({"proto": "udp", "ip_source": ip, "port_dst": 5060})
else:
    print(f"El archivo '{whitelist_sip_file}' no existe. No se cargaron reglas adicionales.")
```

```
# Configuración de reglas predeterminadas
default_allowed_rules = [
    {"proto": "tcp", "ip_source": "172.20.0.100",  "port_dst": 22},
]
# Cargar reglas predeterminadas en el mapa allowed_rules
for rule in default_allowed_rules:
    ip_src = ip_to_u32_inverted(rule["ip_source"])
    port_dst = ctypes.c_uint16(socket.htons(rule["port_dst"]))
    proto = ctypes.c_uint8(PROTO_MAP[rule["proto"].lower()])
    key = allowed_rules3.Key(proto, ip_src, port_dst)
    allowed_rules3[key] = ctypes.c_uint32(1)
```

```
# Endpoint para añadir regla de bloqueo
@app.route('/add_rule_block', methods=['POST'])
def add_rule_block():
    data = request.get_json()
    ip_src = data.get("ip_source")
    port_dst = data.get("port_dst")
    proto_str = data.get("proto", "").lower()  # Protocolo en minúscula

    try:
        # Caso 1: Solo protocolo especificado
        if proto_str and not ip_src and not port_dst:
            if proto_str not in PROTO_MAP:
```

```
            return jsonify({"error": "Protocolo inválido"}), 400
        key = ctypes.c_uint8(PROTO_MAP[proto_str])
        blocked_rules1[key] = ctypes.c_uint32(1)
        return jsonify({"message": f"Regla añadida: bloquear protocolo {proto_str}"}), 200

    # Caso 2: Solo IP fuente especificada
    elif ip_src and not port_dst and not proto_str:
        ip_src_u32 = ip_to_u32_inverted(ip_src)
        key = ctypes.c_uint32(ip_src_u32)
        blocked_rules2[key] = ctypes.c_uint32(1)
        return jsonify({"message": f"Regla añadida: bloquear IP de origen {ip_src}"}), 200

    # Caso 3: IP, puerto y protocolo especificados (original)
    elif ip_src and port_dst and proto_str:
        if proto_str not in PROTO_MAP:
            return jsonify({"error": "Protocolo inválido"}), 400
        key = blocked_rules3.Key(
            ctypes.c_uint8(PROTO_MAP[proto_str]),
            ip_to_u32_inverted(ip_src),
            ctypes.c_uint16(socket.htons(port_dst))
        )
        blocked_rules3[key] = ctypes.c_uint32(1)
        return jsonify({"message": f"Regla añadida: bloquear IP {ip_src} hacia puerto {port_dst} para
{proto_str}"}), 200

    # Error: combinación inválida de campos
    else:
        return jsonify({"error": "Campos incompletos o inválidos. Indique SOLO protocolo, SOLO IP
de origen o ambos junto con puerto destino. Mayor flexibilidad en futuras versiones..."}), 400

  except Exception as e:
    return jsonify({"error": str(e)}), 500

# Endpoint para añadir regla de paso
@app.route('/add_rule_pass', methods=['POST'])
def add_rule_pass():
  data = request.get_json()
  ip_src = data.get("ip_source")  # IP de origen
  port_dst = data.get("port_dst")  # Puerto de destino
  proto_str = data.get("proto", "").lower()  # Protocolo en minúscula

  try:
    # Caso 1: Solo protocolo especificado
    if proto_str and not ip_src and not port_dst:
        if proto_str not in PROTO_MAP:
            return jsonify({"error": "Protocolo inválido"}), 400
        key = ctypes.c_uint8(PROTO_MAP[proto_str])
        allowed_rules1[key] = ctypes.c_uint32(1)
```

```
            return jsonify({"message": f"Regla añadida: permitir protocolo {proto_str}"}), 200

        # Caso 2: Solo IP fuente especificada
        elif ip_src and not port_dst and not proto_str:
            ip_src_u32 = ip_to_u32_inverted(ip_src)
            key = ctypes.c_uint32(ip_src_u32)
            allowed_rules2[key] = ctypes.c_uint32(1)
            return jsonify({"message": f"Regla añadida: permitir IP de origen {ip_src}"}), 200

        # Caso 3: IP, puerto y protocolo especificados (original)
        elif ip_src and port_dst and proto_str:
            if proto_str not in PROTO_MAP:
                return jsonify({"error": "Protocolo inválido"}), 400
            key = allowed_rules3.Key(
                ctypes.c_uint8(PROTO_MAP[proto_str]),
                ip_to_u32_inverted(ip_src),
                ctypes.c_uint16(socket.htons(port_dst))
            )
            allowed_rules3[key] = ctypes.c_uint32(1)
            return jsonify({"message": f"Regla añadida: permitir IP {ip_src} hacia puerto {port_dst} para
{proto_str}"}), 200

        # Error: combinación inválida de campos
        else:
            return jsonify({"error": "Campos incompletos o inválidos. Indique SOLO protocolo, SOLO IP
de origen o ambos junto con puerto destino. Mayor flexibilidad en futuras versiones..."}), 400

    except Exception as e:
        return jsonify({"error": str(e)}), 500


# Endpoint para listar reglas activas
@app.route('/list_rules', methods=['GET'])
def list_rules():
    try:
        allowed = []
        for key, value in allowed_rules1.items():
            # Convertir la IP invertida a formato legible
            proto = [k for k, v in PROTO_MAP.items() if v == key.proto][0]  # Convertir proto numérico
a string
            allowed.append({"proto": proto, "ip_source": "0.0.0.0", "port_dst": 0, "action": "allow"})
        for key, value in allowed_rules2.items():
            # Convertir la IP invertida a formato legible
            ip_src_bytes = struct.pack("!I", key.ip_src)
            ip_src = socket.inet_ntoa(ip_src_bytes[::-1])  # Deshacer inversión de bytes
            allowed.append({"proto": "all", "ip_source": ip_src, "port_dst": 0, "action": "allow"})
        for key, value in allowed_rules3.items():
            # Convertir la IP invertida a formato legible
```

```
            ip_src_bytes = struct.pack("!I", key.ip_src)
            ip_src = socket.inet_ntoa(ip_src_bytes[::-1])  # Deshacer inversión de bytes
            port_dst = socket.ntohs(key.port_dst) # Convertir el puerto a formato host
            proto = [k for k, v in PROTO_MAP.items() if v == key.proto][0]  # Convertir proto numérico
a string
            allowed.append({"proto": proto, "ip_source": ip_src, "port_dst": port_dst, "action": "allow"})

        blocked = []
        for key, value in blocked_rules1.items():
            # Convertir la IP invertida a formato legible
            proto = [k for k, v in PROTO_MAP.items() if v == key.proto][0]  # Convertir proto numérico
a string
            blocked.append({"proto": proto, "ip_source": "0.0.0.0", "port_dst": 0, "action": "block"})
        for key, value in blocked_rules2.items():
            # Convertir la IP invertida a formato legible
            ip_src_bytes = struct.pack("!I", key.ip_src)
            ip_src = socket.inet_ntoa(ip_src_bytes[::-1])  # Deshacer inversión de bytes
            blocked.append({"proto": "all", "ip_source": ip_src, "port_dst": 0, "action": "block"})
        for key, value in blocked_rules3.items():
            # Convertir la IP invertida a formato legible
            ip_src_bytes = struct.pack("!I", key.ip_src)
            ip_src = socket.inet_ntoa(ip_src_bytes[::-1])  # Deshacer inversión de bytes
            port_dst = socket.ntohs(key.port_dst) # Convertir el puerto a formato host
            proto = [k for k, v in PROTO_MAP.items() if v == key.proto][0]  # Convertir proto numérico
a string
            blocked.append({"proto": proto, "ip_source": ip_src, "port_dst": port_dst, "action": "block"})

        return jsonify({"rules": {"blocked": blocked, "allowed": allowed}}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500

# Endpoint para eliminar reglas
@app.route('/delete_rule', methods=['POST'])
def delete_rule():
    try:
        # Recibir datos del cliente
        data = request.get_json()
        action = data.get("action")
        ip_source = data.get("ip_source")
        port_dst = data.get("port_dst")
        proto_str = data.get("proto", "").lower()

        print(f"Datos recibidos en /delete_rule: {data}")

        # Validar acción
        if action not in ["block", "allow"]:
            return jsonify({"error": "Acción inválida. Use 'block' o 'allow'"}), 400
```

```
        # Seleccionar la tabla según la acción
        target_table_1 = blocked_rules1 if action == "block" else allowed_rules1
        target_table_2 = blocked_rules2 if action == "block" else allowed_rules2
        target_table_3 = blocked_rules3 if action == "block" else allowed_rules3

        # Caso 1: Eliminar por IP + Puerto + Protocolo
        if ip_source and port_dst and proto_str:
            key = target_table_3.Key(
                ctypes.c_uint8(PROTO_MAP[proto_str]),
                ip_to_u32_inverted(ip_source),
                ctypes.c_uint16(socket.htons(port_dst))
            )
            del target_table_3[key]
            return jsonify({"message": f"Regla eliminada: {action} IP {ip_source} puerto {port_dst} protocolo {proto_str}"}), 200

        # Caso 2: Eliminar por solo IP
        elif ip_source and not port_dst and proto_str == "all":
            key = ctypes.c_uint32(ip_to_u32_inverted(ip_source))
            del target_table_2[key]
            return jsonify({"message": f"Regla eliminada: {action} IP {ip_source}"}), 200

        # Caso 3: Eliminar por solo Protocolo
        elif proto_str and ip_source == "0.0.0.0" and not port_dst:
            key = ctypes.c_uint8(PROTO_MAP[proto_str])
            del target_table_1[key]
            return jsonify({"message": f"Regla eliminada: {action} protocolo {proto_str}"}), 200

        # Si no coincide con ninguno de los casos válidos
        return jsonify({"error": "Campos incompletos o inválidos"}), 400

    except KeyError:
        return jsonify({"error": "Regla no encontrada"}), 404
    except Exception as e:
        print(f"Error en /delete_rule: {str(e)}")
        return jsonify({"error": str(e)}), 500
```

```
if __name__ == '__main__':
    try:
        print(f"Programa eBPF asociado a la interfaz {INTERFACE}. Presiona Ctrl+C para salir.")
        app.run(host='0.0.0.0', port=5027, ssl_context=('cert_back.pem', 'key_back.pem'))
    except KeyboardInterrupt:
        cleanup(None, None)
```

Sección 3. Programa backend para el eBPF Master: “app.py”

```
from flask import Flask, render_template, request, jsonify, redirect, url_for, session
import requests
import psycopg2
from functools import wraps
import bcrypt
import socket
from jsonschema import validate, ValidationError
```

```
schema = {
    "$schema": "http://json-schema.org/draft-07/schema#",
    "type": "object",
    "properties": {
        "action": {
            "type": "string",
            "enum": ["block", "allow"]
        },
        "ip_source": {
            "type": "string",
            "format": "ipv4"
        },
        "port_dst": {
            "type": "integer",
            "minimum": 1,
            "maximum": 65535
        },
        "proto": {
            "type": "string",
            "enum": ["tcp", "udp", "icmp", "all"]
        }
    },
    "additionalProperties": False
}
```

```
@app.route('/')
@login_required
def index():
    return render_template('index.html')

# Decorador para proteger rutas - No aplicado por ahora
def login_required(f):
    @wraps(f)
    def decorated_function(*args, **kwargs):
        if 'user_id' not in session:
```

```
            return redirect(url_for('login_page'))
        return f(*args, **kwargs)
    return decorated_function

# Ruta para el formulario de login
@app.route('/login', methods=['GET', 'POST'])
def login_page():
    if request.method == 'POST':
        username = request.form.get('username')
        password = request.form.get('password')

        if not username or not password:
            return render_template('login.html', error="Faltan credenciales")

        try:
            conn = get_db_connection()
            cur = conn.cursor()

            # Consultar la base de datos para obtener el hash almacenado
            cur.execute("SELECT password FROM users WHERE username = %s", (username,))
            result = cur.fetchone()

            if result is not None:
                stored_password = result[0]
                if bcrypt.checkpw(password.encode('utf-8'), stored_password.encode('utf-8')):
                    print("OK")
                    #session['user_id'] = user_id  # Set user ID in session (assuming you have a user_id
retrieved)
                    print("Redirecting to index...")
                    return redirect(url_for('index2'))
            else:
                return render_template('login.html', error="Usuario o contraseña incorrectos")

        except Exception as e:
            return render_template('login.html', error=f"Error: {str(e)}")
        finally:
            if conn:
                conn.close()

    # Si es una petición GET, renderizar el formulario de login
    return render_template('login.html')

@app.route('/login_ok')
def index2():
    return render_template('index.html')
```

```
@app.route('/list_rules', methods=['GET'])
def list_rules():
    try:
        # Llamada al endpoint de la API
        response = requests.get(f"{API_BASE_URL}/list_rules", verify='cert_back.pem')
        rules = response.json()
        return jsonify(rules)  # Devuelve las reglas en formato JSON
    except Exception as e:
        return jsonify({"error": str(e)}), 500

@app.route('/add_rule', methods=['POST'])
def add_rule():
    try:
        # Recibir datos del formulario
        data = request.get_json()
        action = data.get("action")  # Puede ser "block" o "allow"
        ip_source = data.get("ip_source")
        port_dst = data.get("port_dst")
        proto = data.get("proto")  # Protocolo opcional

        # Validar acción
        if action not in ["block", "allow"]:
            return jsonify({"error": "Acción inválida. Use 'block' o 'allow'"}), 400

        # Validar datos opcionales
        if port_dst:
            try:
                port_dst = int(port_dst)  # Convertir el puerto a un entero si se proporciona
            except ValueError:
                return jsonify({"error": "El puerto debe ser un número entero"}), 400

        if proto:
            valid_protos = ["tcp", "udp", "icmp"]
            if proto.lower() not in valid_protos:
                return jsonify({"error": f"Protocolo inválido. Use uno de {valid_protos}"}), 400

        # Seleccionar el endpoint correcto
        endpoint = "/add_rule_block" if action == "block" else "/add_rule_pass"

        # Construir el payload dinámicamente según los valores presentes
        payload = {key: value for key, value in {
            "ip_source": ip_source,
            "port_dst": port_dst,
            "proto": proto.lower() if proto else None
        }.items() if value is not None}

        try:
            validate(instance=payload, schema=schema)
```

```
        except ValidationError as e:
            print("Error de validación:", e.message)
            # Manejar el error, por ejemplo, abortar la solicitud
        else:
            # Enviar la solicitud POST si la validación es exitosa
            response = requests.post(f"{API_BASE_URL}{endpoint}", json=payload,
verify='cert_back.pem')
        # Devolver la respuesta de la API

        return jsonify(response.json())
    except Exception as e:
        return jsonify({"error": str(e)}), 500


@app.route('/delete_rule', methods=['POST'])
def delete_rule():
    try:
        # Obtener los datos enviados en el cuerpo de la solicitud
        data = request.get_json()
        action = data.get("action")  # "block" o "allow"
        ip_source = data.get("ip_source")
        port_dst = data.get("port_dst")
        proto = data.get("proto")  # Puede ser opcional

        # Validar acción
        if action not in ["block", "allow"]:
            return jsonify({"error": "Acción inválida. Use 'block' o 'allow'"}), 400

        # Construir el payload dinámicamente según los valores presentes
        payload = {key: value for key, value in {
            "action": action,  # Se incluye la acción en el payload
            "ip_source": ip_source,
            "port_dst": port_dst,
            "proto": proto.lower() if proto else None
        }.items() if value is not None}

        try:
            validate(instance=payload, schema=schema_del)
        except ValidationError as e:
            print("Error de validación:", e.message)
            # Manejar el error, por ejemplo, abortar la solicitud
        else:
            # Registrar el payload enviado
            print(f"Enviando payload a {API_BASE_URL}/delete_rule: {payload}")
            # Enviar la solicitud POST si la validación es exitosa
            response = requests.post(f"{API_BASE_URL}/delete_rule", json=payload,
verify='cert_back.pem')
```

```
        # Manejar respuesta del backend
        try:
            response_data = response.json()  # Intentar parsear la respuesta JSON
        except ValueError:
            # Si no es JSON, devolver el texto plano del error
            return jsonify({"error": f"Error en el backend: {response.text}"}), response.status_code

        # Si la respuesta es exitosa, retornar el mensaje del backend
        if response.status_code == 200:
            return jsonify({"message": response_data.get("message", "Regla eliminada correctamente")})
        else:
            # Si hay un error, retornar el mensaje del backend
            return jsonify({"error": response_data.get("error", "Error desconocido")}), response.status_code

    except Exception as e:
        # Registrar errores en la aplicación
        print(f"Error en /delete_rule: {str(e)}")
        return jsonify({"error": str(e)}), 500

@app.route('/get_agent_hostname', methods=['GET'])
def get_agent_hostname():
    try:
        hostname = socket.gethostname()  # Obtiene el hostname del sistema
        return jsonify({"hostname": 'firewall-ebpf-test.bts.io'}), 200
    except Exception as e:
        return jsonify({"error": str(e)}), 500

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5028, ssl_context=('cert_front.pem', 'key_front.pem'))
```

Sección 4. Gestión web

Programa register.py

```
import bcrypt
import psycopg2
import argparse

# Configuración de la base de datos
DB_CONFIG = {
    'dbname': 'firewall_users',
    'user': ‘xxxxxxxx’,
    'password':,
    'host': 'localhost',
    'port': '5432'
}
```

```
def hash_password(password):
    """
    Genera un hash seguro para la contraseña proporcionada.
    """
    return bcrypt.hashpw(password.encode('utf-8'), bcrypt.gensalt()).decode('utf-8')

def insert_user(username, password):
    """
    Inserta un usuario en la base de datos con su contraseña hasheada.
    """
    try:
        # Conexión a la base de datos
        conn = psycopg2.connect(**DB_CONFIG)
        cur = conn.cursor()

        # Hashear la contraseña
        hashed_password = hash_password(password)

        # Insertar usuario en la base de datos
        cur.execute("INSERT INTO users (username, password) VALUES (%s, %s)", (username,
hashed_password))
        conn.commit()

        print(f"Usuario '{username}' insertado correctamente.")
    except Exception as e:
        print(f"Error al insertar usuario: {e}")
    finally:
        if conn:
            cur.close()
            conn.close()

if __name__ == "__main__":
    # Configuración del argumento de línea de comandos
    parser = argparse.ArgumentParser(description="Añadir un usuario con contraseña hasheada a la
base de datos.")
    parser.add_argument("username", help="Nombre del usuario a añadir.")
    parser.add_argument("password", help="Contraseña del usuario.")

    # Parsear argumentos
    args = parser.parse_args()

    # Llamar a la función para insertar el usuario
    insert_user(args.username, args.password)
```

ANEXO F. Aplicación de políticas de seguridad y puesta en marcha del sistema.

En este anexo se detalla la aplicación de distintas medidas de seguridad involucradas en el proyecto. Además, se desarrolla la puesta en marcha del sistema completo, una vez está listo para ello. Todo ello, siguiendo con lo mencionado en los *Apartados 4.5 y 4.6.*

Sistema de login

El sistema de login implementado en la web de gestión de los agentes tiene como objetivo principal restringir el acceso a la administración centralizada únicamente a personas autorizadas. La correcta gestión de los roles de los trabajadores es esencial en una empresa como BTS, especialmente cuando se trata de un punto de control centralizado con un alto nivel de poder de gestión.

La implementación del sistema de autenticación comienza con una ventana de login que se presenta antes de permitir el acceso a la interfaz principal del Master eBPF. Para lograrlo, se incluye en el programa *app.py* un endpoint específico (*/login*) que cumple dos funciones: primero, mostrar al usuario un **formulario de login**, y segundo, **verificar sus credenciales contra una base de datos PostgreSQL** alojada en el propio Master eBPF. [Código en el ANEXO E – Sección 3].

El usuario es redireccionado a este endpoint cuando accede a la ruta principal de la web (https://<IP_eBPF_Master:5028/), y se accede a la interfaz de *login.html*.

Para registrar usuarios en el sistema, y como medida provisional durante la fase de pruebas, se ha desarrollado un script en Python llamado *register.py*. [Código en el ANEXO E – Sección 4]. Este script permite guardar pares usuario-contraseña en la base de datos local. Con el objetivo de garantizar la seguridad, el script **almacena un HASH de la contraseña** en lugar de la contraseña en texto plano, reforzando así la **protección de los datos al registrar nuevos usuarios**. En el momento en el que el usuario quiere autenticarse, el propio script *app.py* es el encargado de **desencriptar la contraseña almacenada** y autenticarlo si corresponde.

La base de datos utilizada para almacenar los usuarios y sus contraseñas es gestionada mediante PostgreSQL, y su estructura se muestra en el ANEXO C. Tal como se puede observar, las contraseñas no se almacenan en texto plano, sino que se guardan de forma codificada para garantizar una mayor seguridad.

Certificados TLS/SSL

En el sistema desarrollado, la seguridad en la comunicación entre componentes se logra mediante el uso de **certificados TLS autofirmados** (útiles para entornos de pruebas). Estos certificados garantizan que la comunicación HTTP sea segura, protegiendo la **confidencialidad e integridad** de los datos transmitidos. Tanto el programa del backend eBPF

add_rule.py como el backend de la web *app.py*, se ejecutan incluyendo estos certificados en el momento de abrir el puerto de conexión. Se utiliza un certificado (*cert*) y una clave privada (*key*).

```
app.run(host='0.0.0.0', port=5027, ssl_context=('cert_back.pem', 'key_back.pem'))
app.run(host='0.0.0.0', port=5028, ssl_context=('cert_front.pem', 'key_front.pem'))
```

Los **certificados** contienen la clave pública del servidor, y se utilizan para verificar la identidad del servidor hacia los clientes (**negociación TLS**). Las **claves privadas** son archivos secretos, y corresponden a la clave pública existente en el certificado. Se utilizan para desencriptar datos cifrados por los clientes.

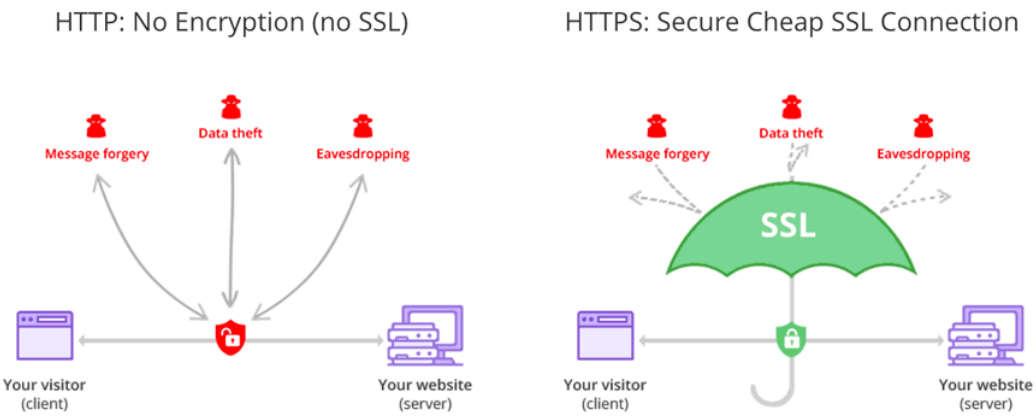


Figura 51. Funcionamiento de SSL/TLS.

- **add_rule.py**: *'cert_back.pem'* y *'key_back.pem'*

Este certificado (*cert*) y la clave privada (*key*) se encargan de proteger el backend que corre en el puerto 5027 en los agentes eBPF. Aseguran que peticiones al backend, como */list_rules* estén encriptadas. Las peticiones HTTP provenientes de *app.py* utilizan el archivo *cert.pem* como certificado de confianza.

Ejemplo de petición al backend *add_rule.py*:

```
response = requests.get(f"{API_BASE_URL}/list_rules", verify='cert_back.pem')
```

- **app.py**: *'cert_front.pem'* y *'key_front.pem'*

Este par de certificado y clave privada tienen la función de proteger el acceso a la interfaz de administración del eBPF Master, en el puerto 5028. Garantizan que las conexiones de los usuarios a la web de gestión sean seguras y protegidas contra ataques como *man-in-the-middle*. Permiten que la conexión con el servidor web se realice mediante **HTTPS**.

JSON Schemas

Como se mencionó en el *Apartado 4.3*, los datos enviados en las peticiones son validados mediante **JSON Schemas**, los cuales definen tanto el **tipo como el formato permitido para los datos.** Estos esquemas se diseñan y configuran específicamente para garantizar que las peticiones cumplan con los requisitos establecidos.

Su uso contribuye a la seguridad al establecer restricciones claras sobre el tipo y formato de los datos que pueden enviarse en las peticiones. Esto reduce el riesgo de ataques como la **inyección de datos maliciosos**, ya que cualquier información que no cumpla con las especificaciones es rechazada automáticamente. Además, los schemas minimizan errores y aseguran que los datos procesados sean fiables, evitando comportamientos inesperados en el sistema.

Puesta en marcha del sistema.

En el Master, se debe ejecutar el programa *app.py*, que habilita el puerto necesario para el acceso a la interfaz web. En los agentes, es fundamental iniciar el programa *add_rule.py*, encargado de activar el Firewall eBPF en la máquina correspondiente. Una vez **ambos componentes están activos**, el sistema está completamente operativo y puede gestionarse desde el Master.

Durante el desarrollo en el entorno de pruebas, los programas se ejecutan manualmente mediante comandos en el momento de su aplicación (*Figura 52*). Sin embargo, para la implementación en producción, se contempla la **creación de un servicio que permita que los programas estén corriendo de manera continua en las máquinas**. Esto garantizaría que el firewall permanezca siempre accesible y en pleno funcionamiento.

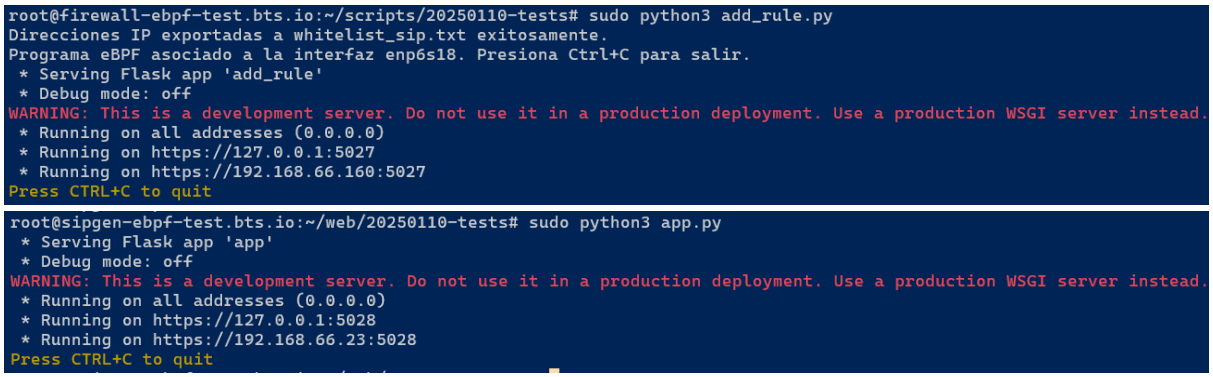


Figura 52. Ejecución de los programas involucrados

Es fundamental asegurarse de que los puertos abiertos estén permitidos dentro de las políticas del firewall, tanto para el acceso desde el Master como desde los agentes. De no ser así, las conexiones serán rechazadas, lo que impediría la comunicación efectiva entre los diferentes componentes del sistema.

ANEXO G. Pruebas de funcionamiento generales.

Funcionalidades del Firewall

Esta sección se centra en verificar el correcto funcionamiento de las diversas funcionalidades del Firewall eBPF descritas a lo largo de la memoria. Aunque estas no intervienen directamente en la toma de decisiones sobre el tráfico ni en la ejecución de funciones eBPF, desempeñan un papel fundamental para garantizar el funcionamiento integral del sistema, proporcionando capacidades esenciales para el firewall.

- **Sistema de Login y adición de usuarios**

Como ya se ha comentado en el *Capítulo 4*, el sistema utiliza un script en Python (*register.py*) para añadir usuarios permitidos en el sistema, utilizando una base de datos PostgreSQL localizada en el eBPF Master. El uso es el siguiente para correr el script:
 # sudo python3 register.py <username> <password>.

Como ejemplo, añadimos un usuario en la máquina *sipgen-ebpf*:
sipgen-ebpf# ***sudo python3 register.py usuario_tfm usuario_tfm***
Usuario 'usuario_tfm' insertado correctamente.

Al consultar en la base de datos, veremos cómo el usuario se ha añadido junto a su contraseña codificada. El usuario ya podrá acceder a la gestión de los agentes eBPF con su contraseña.

```
firewall_users=# select * from users;
 id |  username   |                          password
----+-------------+------------------------------------------------------------
  1 | cirigoyen   | $2b$12$6l6Bqc.fr3JQji2t6ACKreZpOTHcqiA7FUxr8iBqyc9Sg2J.bmavS
  2 | jlazaro     | $2b$12$dUssBPId0QL.cc.M0VOtLeR1lVxfHLOERHzD9NBwYnFS9Pog4bHY6
  3 | usuario_tfm | $2b$12$o3jSAFQ77NyJMHS1hOHZwOdoEXalxpPj40DspKBJy7rumfzTkbCsC
(3 rows)
```

Figura 53. Nuevo usuario en la base de datos.

- **Reglas por defecto y lectura de base de datos**

Otra funcionalidad fundamental para el correcto funcionamiento es la lectura de reglas por defecto de permiso y de IPs de una base de datos, para conformar las “Whitelist”.

Respecto a las reglas por defecto, pueden configurarse manualmente en el código *add_rule.py* (ver ANEXO E), mediante la estructura *default_allowed_rules*. Como ejemplo, vamos a preparar una regla por defecto concreta para permitir el tráfico UDP de la IP 100.100.100.100 hacia el puerto 22:

```
# Configuración de reglas predeterminadas
default_allowed_rules = [
    {"proto": "udp", "ip_source": "100.100.100.100",  "port_dst": 22},
]
```

Figura 54. Regla por defecto de prueba.

Por otro lado, se va a probar la lectura de IPs para conformar la *whitelist* de tráfico SIP desde una base de datos. Añadimos las siguientes tres IPs a la base de datos, como ejemplo para permitir el tráfico SIP (Puerto 5060, UDP):

```
sip_whitelist=# select * from direcciones_ip ;
   ip_address
-----------------
 192.168.67.7
 100.100.100.100
 199.20.20.20
(3 rows)
```

Figura 55. Consulta de Whitelist SIP en la base de datos.

Ahora, una vez el Firewall eBPF es activado, comprobamos la aparición de la regla concreta para permitir el tráfico por defecto, junto a las tres reglas de tráfico SIP que han leído las IPs de la base de datos:

Reglas Activas

Acción	IP Fuente	Puerto Destino	Protocolo	Acciones
Permitir	199.20.20.20	5060	UDP	Delete
Permitir	100.100.100.100	5060	UDP	Delete
Permitir	192.168.67.7	5060	UDP	Delete
Permitir	100.100.100.100	22	UDP	Delete

Figura 56. Carga de reglas por defecto y lectura de Whitelist SIP

Pruebas básicas iniciales con tráfico

Con el sistema del Firewall eBPF completamente operativo y preparado para gestionar el tráfico en el agente, se llevan a cabo las primeras pruebas para verificar su funcionamiento. Estas pruebas iniciales tienen como objetivo principal confirmar la capacidad del Firewall eBPF para bloquear tráfico, comenzando con el análisis de situaciones de tráfico específicas y simples.

- **Bloqueo de PING (protocolo ICMP)**

En esta primera prueba se añadió una regla el Firewall eBPF en el agente (máquina *firewall-ebpf*) para bloquear todo el tráfico ICMP (protocolo usado por PING). Esto, por defecto está permitido en el Firewall Linux convencional con *iptables*.

Situación inicial, mandando 1 paquete PING desde el Master hacia el agente:

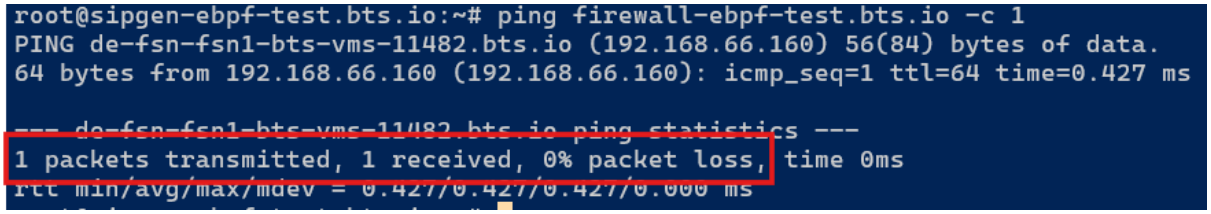


Figura 57. PING bloqueado.

Inclusión de la nueva regla para bloquear el protocolo ICMP:

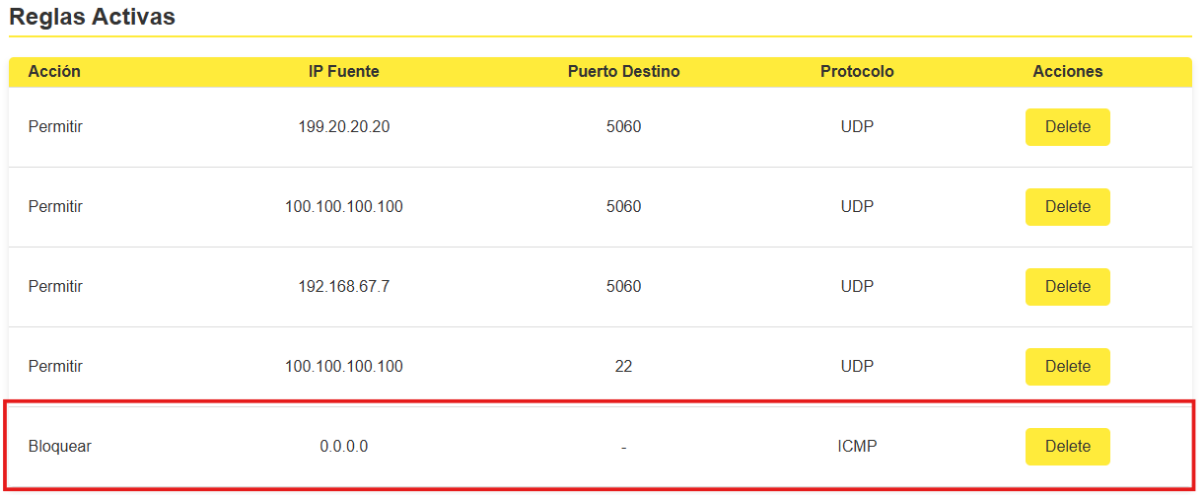


Figura 58. Nueva regla para ICMP.

Se vuelve a lanzar el mismo test, y efectivamente, el PING es bloqueado:

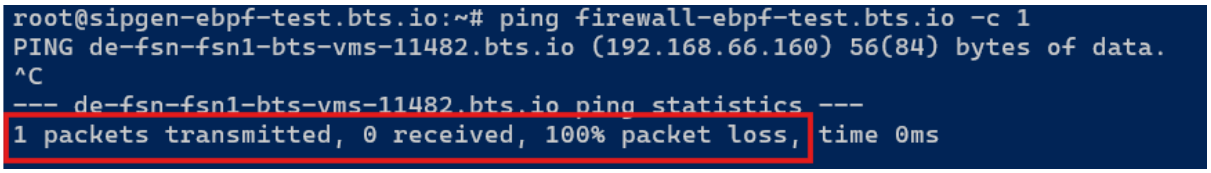


Figura 59. PING no bloqueado.

- **Bloqueo de SSH**

Con el fin de comprobar el funcionamiento de reglas que utilicen los tres campos a completar en el formulario del firewall, se repite la situación anterior, pero esta vez lanzando una conexión SSH desde el Master hacia el agente.

Se añade la regla para bloquear el tráfico SSH (Puerto 22, protocolo TCP) proveniente de la máquina *sipgen-ebpf*:

Reglas Activas

Acción	IP Fuente	Puerto Destino	Protocolo	Acciones
Permitir	199.20.20.20	5060	UDP	Delete
Permitir	100.100.100.100	5060	UDP	Delete
Permitir	192.168.67.7	5060	UDP	Delete
Permitir	100.100.100.100	22	UDP	Delete
Bloquear	192.168.66.23	22	TCP	Delete

Figura 60. Nueva regla de bloqueo.

Y se comprueba que no se obtiene respuesta al intentar la conexión ya que está siendo bloqueada:

```
root@sipgen-ebpf-test.bts.io:~# ssh firewall-ebpf-test.bts.io
```

Figura 61. SSH bloqueado.

ANEXO H. Complementos a las pruebas de carga.

En este anexo se muestran diversos complementos a lo presentado en el *Apartado 5.2*, referente a las pruebas de funcionamiento y de carga del sistema.

Escenario 1: Carga media

Tabla de resultados con todas las pruebas realizadas, a partir de las cuales se obtienen los valores promedio presentados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists) Prueba 1	2.78 GB	1.28%	8.91%	**7.63%**
IPTABLES (4 Whitelists) Prueba 2	2.78 GB	0.82%	8.35%	**7.53%**
IPTABLES (2 Whitelist) P1	2.78 GB	1.24%	8.05%	**6.81%**
IPTABLES (2 Whitelist) P2	2.78 GB	1.28%	8.28%	**7.00%**
Firewall eBPF P1	2.79 GB	1.29%	5.88%	**4.59%**
Firewall eBPF P2	2.79 GB	1.23%	5.77%	**4.54%**

Estado de *iptables* y sus reglas configuradas para cada situación. En las dos primeras columnas, la cantidad de tráfico bloqueado:

- *Iptables* - 4 *whitelists* aplicadas antes de bloquear el tráfico:

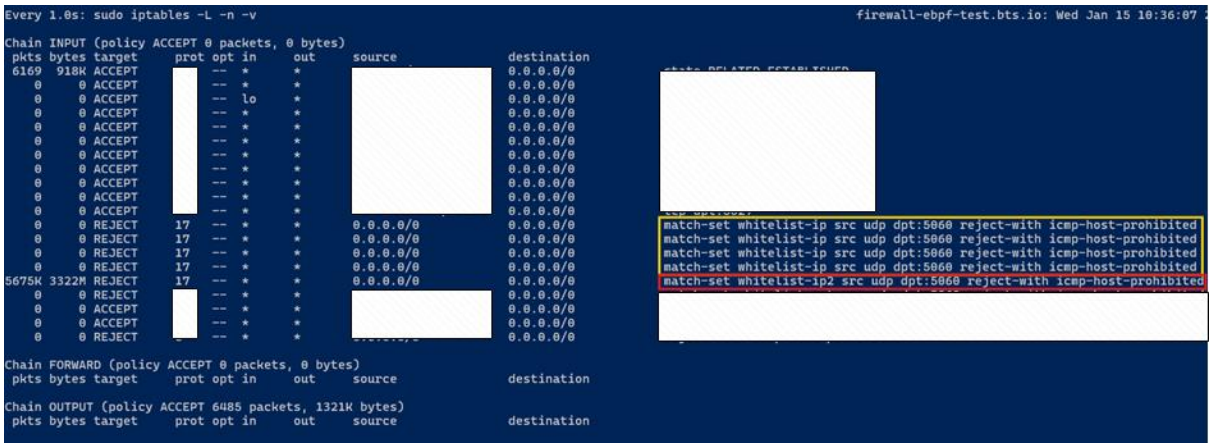


Figura 62. Iptables con 4 whitelists.

- *Iptables* - 2 *whitelists* aplicadas antes de bloquear el tráfico:

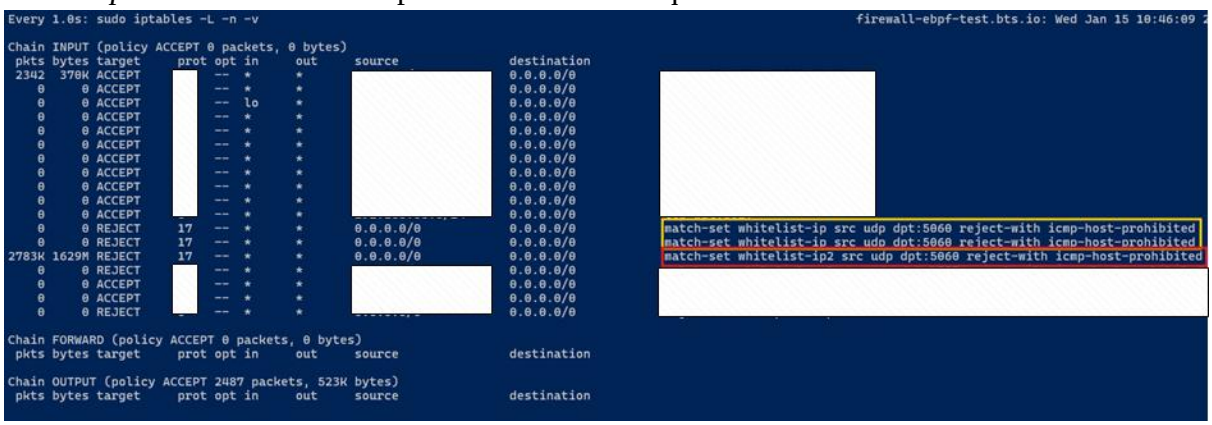


Figura 63. Iptables con 2 whitelists.

- Bloqueando con eBPF:

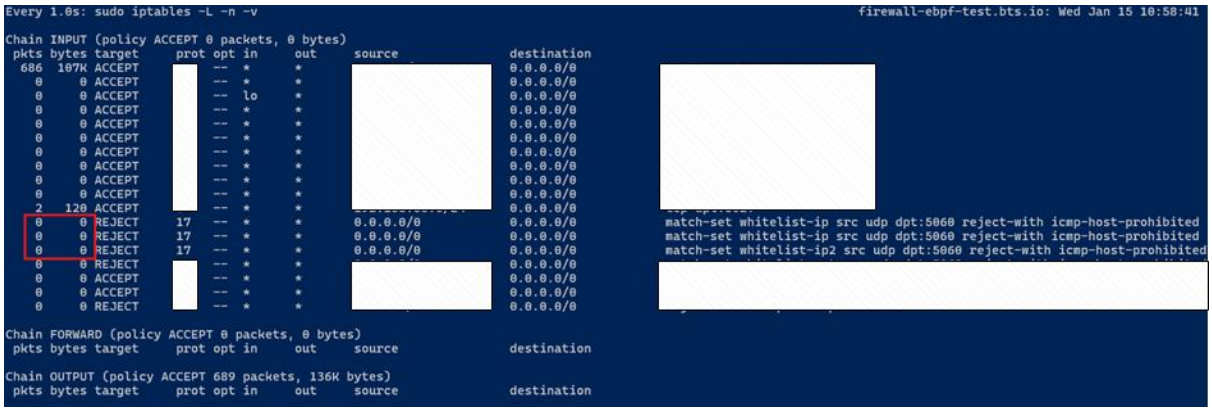


Figura 64. Iptables con eBPF bloqueando el tráfico.

Escenario 2: Carga alta no prolongada

Tabla de resultados con todas las pruebas realizadas, a partir de las cuales se obtienen los valores promedio presentados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists) Prueba 1	2.82 GB	1.30%	17.66%	**16.36%**
IPTABLES (4 Whitelists) Prueba 2	2.82 GB	2.05%	16.76%	**14.71%**
IPTABLES (2 Whitelist) P1	2.83 GB	1.32%	14.96%	**13.64%**
IPTABLES (2 Whitelist) P2	2.83 GB	1.27%	16.45%	**15.18%**
Firewall eBPF P1	2.83 GB	1.26%	9.46%	**8.20%**
Firewall eBPF P2	2.83 GB	1.26%	10.17%	**8.91%**

Estado de *iptables* y sus reglas configuradas para cada situación. En las dos primeras columnas, la cantidad de tráfico bloqueado:

- *Iptables* - 4 *whitelists* aplicadas antes de bloquear el tráfico:

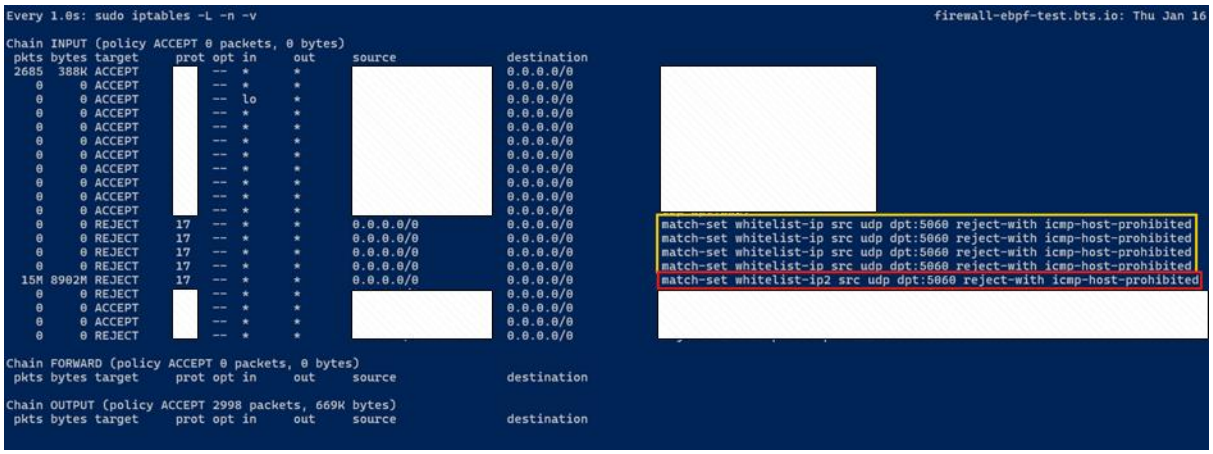


Figura 65. Iptables con 4 whitelists.

- *Iptables* - 2 *whitelists* aplicadas antes de bloquear el tráfico:

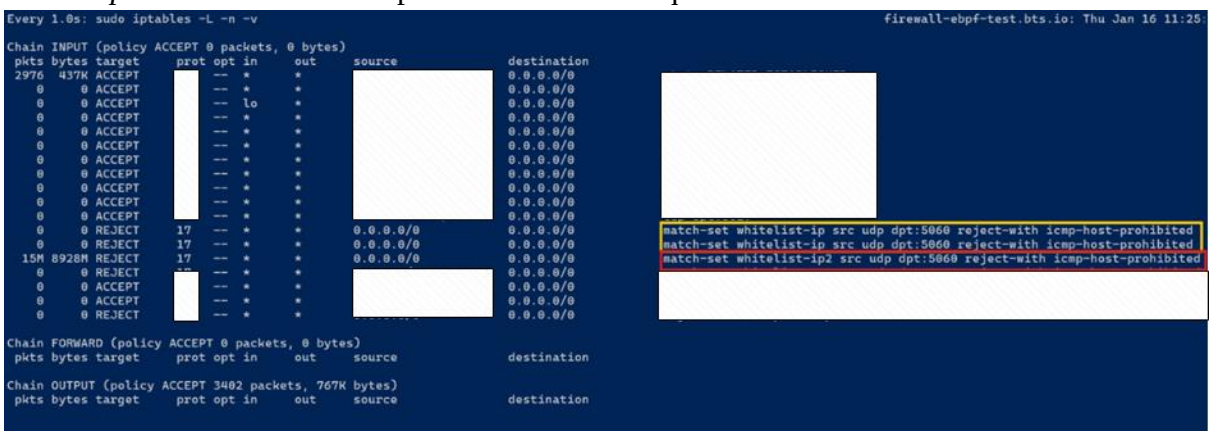


Figura 66. Iptables con 2 whitelists.

- Bloqueando con eBPF:

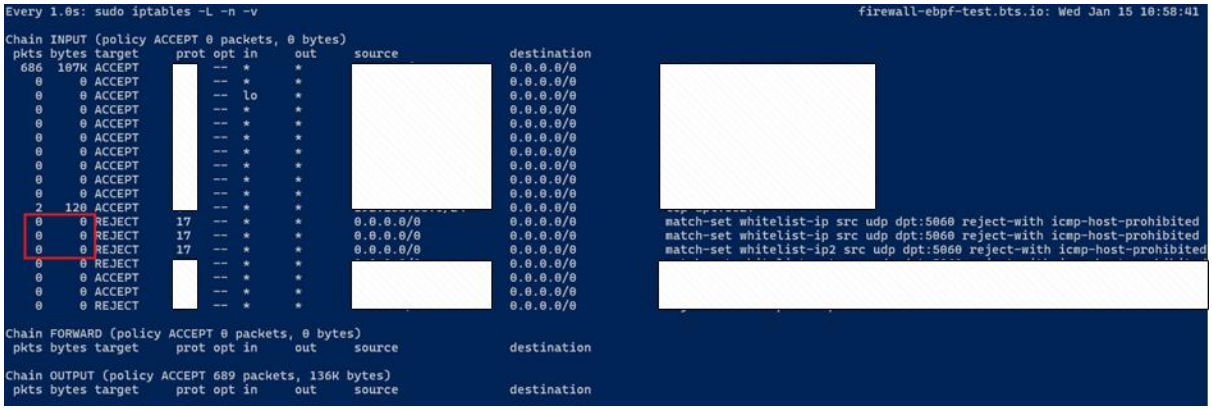


Figura 67. Iptables con eBPF bloqueando el tráfico.

Escenario 3: Carga alta prolongada

Tabla de resultados con todas las pruebas realizadas, a partir de las cuales se obtienen los valores promedio presentados:

Método de bloqueo	**%MEM (max)**	**%CPU (inicial)**	**%CPU (max)**	**%CPU (subida)**
IPTABLES (4 Whitelists) Prueba 1	2.83 GB	1.18%	16.67%	**14.93%**
IPTABLES (4 Whitelists) Prueba 2	2.82 GB	1.36%	17.27%	**15.91%**
IPTABLES (2 Whitelists) P1	2.82 GB	1.16%	15.35%	**14.19%**
IPTABLES (2 Whitelists) P2	2.83 GB	1.23%	15.53%	**14.30%**
IPTABLES (Bloqueo directo) P1	2.83 GB	1.25%	14.54%	**13.29%**
IPTABLES (Bloqueo directo) P2	2.82 GB	1.25%	13.63%	**12.38%**
Firewall eBPF P1	2.83 GB	1.15%	9.71%	**8.56%**
Firewall eBPF P2	2.83 GB	1.21%	9.82%	**8.61%**

Estado de *iptables* y sus reglas configuradas para cada situación. En las dos primeras columnas, la cantidad de tráfico bloqueado:

- *Iptables* - 4 *whitelists* aplicadas antes de bloquear el tráfico:

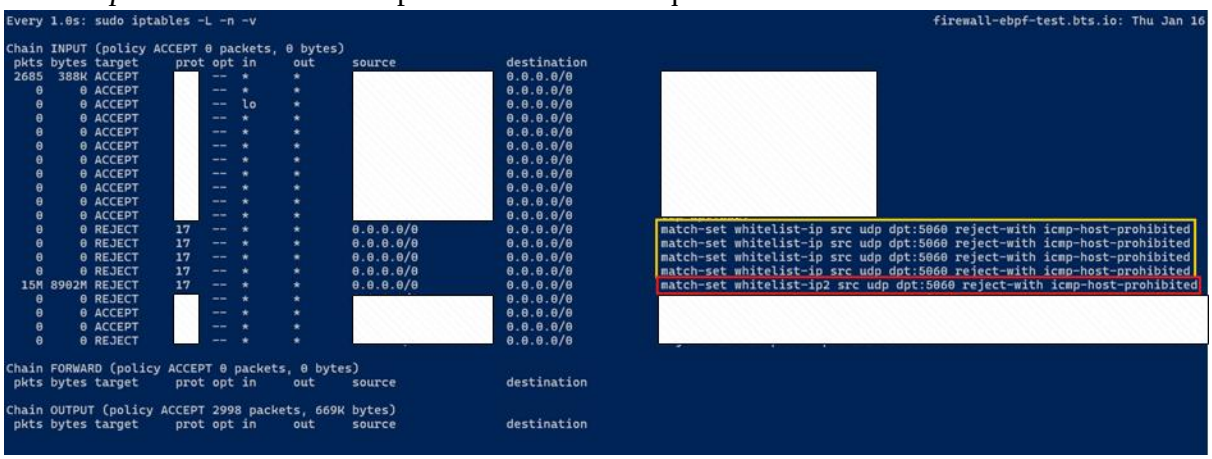


Figura 68. Iptables con 4 Whitelists.

- *Iptables* - 2 *whitelists* aplicadas antes de bloquear el tráfico:

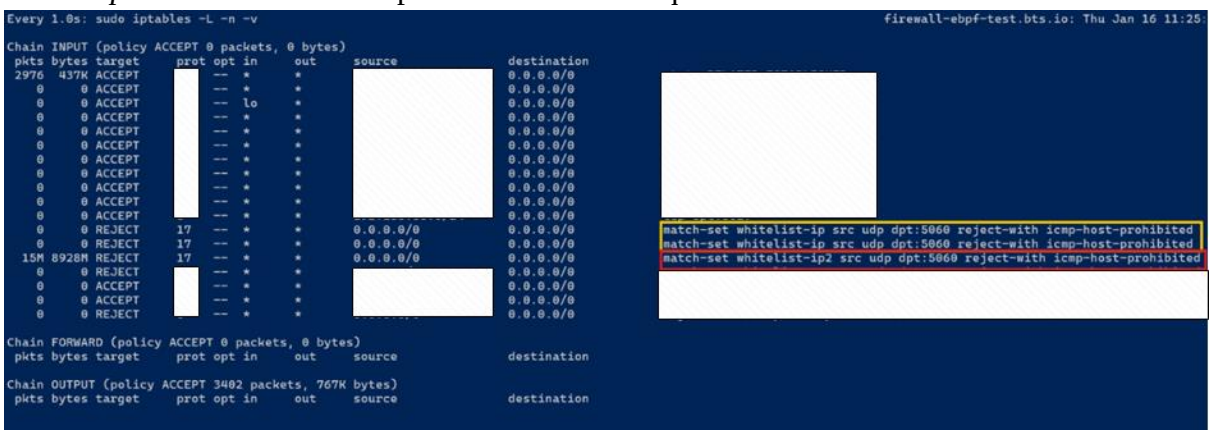


Figura 69. Iptables con 2 whitelists.

- *Iptables* - Bloqueo directo del tráfico:

Figura 70. Iptables bloqueando sin whitelists.

- Bloqueando con eBPF:

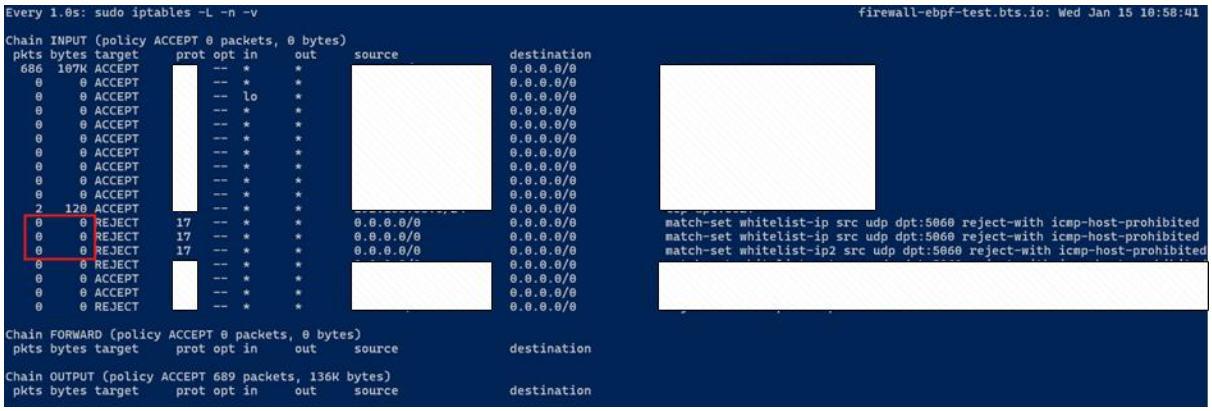
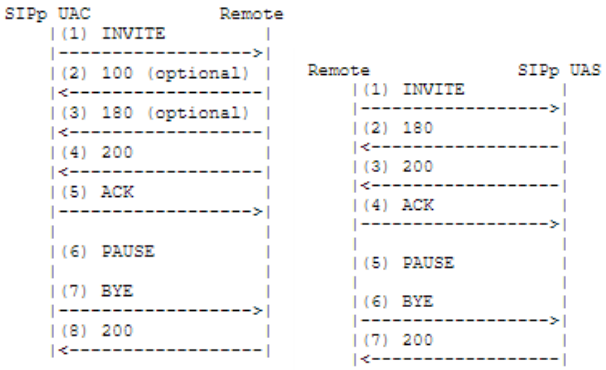


Figura 71. Iptables con eBPF bloqueando el tráfico.

ANEXO I. Uso de la herramienta SIPp.

En este anexo se explica en detalle el funcionamiento de SIPp, y se expone el uso que se ha hecho de la herramienta para las pruebas realizadas.

1. Funcionamiento de escenarios básicos: uas, uac. Control de la aplicación.



uac.xml *uas.xml*

Estos son los dos escenarios básicos que nos proporciona SIPp. funcionan con dos "usuarios": Servidor (*uas*) y Cliente (*uac*).

La máquina que funciona como Servidor queda en escucha de llamadas entrantes para responder con la señalización SIP correspondiente para el establecimiento de ellas.

Como podemos ver, nuestro *uas* espera recibir un mensaje INVITE que provenga de un remoto, y pasar a establecer la llamada. Enviará un mensaje *180* y *200 OK*. Posteriormente, se puede configurar una pausa, que simula la duración de las llamadas. Finalmente espera recibir un *BYE*, respondiendo con un *200 OK* para terminar la llamada.

Por otro lado, el Cliente (nuestro *uac*) funcionará como llamante. Será el encargado de iniciar la señalización SIP que dará lugar a las llamadas. En este caso el *uac* envía los mensajes esperados por el *uas*, y recibe la señalización procedente de él. Será el encargado de enviar el *BYE* y por tanto finalizar la llamada.

La herramienta SIPp se ejecuta desde línea de comandos, escogiendo el escenario que se desea. La máquina que actúa como Cliente (*uac*), deberá indicar la IP del Servidor (*uas*) con el que sedea comunicarse. El funcionamiento básico para que se ponga en marcha la comunicación entre cliente y servidor es el siguiente:

En el cliente: `# ./sipp -sn uac 127.0.0.1`
En el servidor: `# ./sipp -sn uas`

Podemos controlar el funcionamiento de SIPp de forma interactiva, mediante teclas ('hot' keys) que modifican el tráfico generado en cualquier momento. Las hot keys son:

Key	Action
+	Increase the call rate by 1 * rate_scale
*	Increase the call rate by 10 * rate_scale
-	Decrease the call rate by 1 * rate_scale
/	Decrease the call rate by 10 * rate_scale
c	Enter command mode
q	Quit SIPp (after all calls complete, enter a second time to quit immediately)
Q	Quit SIPp immediately
s	Dump screens to the log file (if -trace_screen is passed)
p	Pause traffic
1	Display the scenario screen
2	Display the statistics screen
3	Display the repartition screen
4	Display the variable screen
5	Display the TDM screen
6-9	Display the second through fifth repartition screen.

También disponemos del modo comandos, con el que se puede escribir un comando de una sola línea que indique a SIPp que realice alguna acción. El modo de comando es más versátil que las teclas de acceso rápido, pero toma más tiempo ingresar algunas acciones comunes.

Además para controlar el tráfico se pueden determinar parámetros de inicio, que se especifican a la hora de ejecutar los escenarios. Algunos de los más útiles son:

`-sn:` Usa un escenario por defecto (incluido con el ejecutable de SIPp). Si se omite esta función, el escenario estándar SipStone UAC se ejecuta.

`-sf:` Carga un archivo de escenario XML alternativo a los existentes por defecto. Para aprender más sobre la sintaxis de los escenarios XML se puede utilizar la opción -sd para mostrar escenarios incluidos. Contienen toda la ayuda necesaria.

`-i:` Establece la IP local para las cabeceras 'Contact:','Via:', y 'From:'. Por defecto es la IP primaria del host.

`-d:` Controla la duración de las llamadas. Más precisamente, controla la duración de las instrucciones de 'pausa' del escenario. Su valor por defecto es 0 y su unidad es milisegundos.

`-s:` Establece el username del URI de petición. Por defecto es 'service'.

`-r:` Establece la tasa de llamadas (en llamadas por segundo). El valor por defecto es 10.

`-rp:` Establece el periodo de la tasa de llamadas. Por defecto es 1 segundo y su unidad es milisegundos. Permite realizar n llamadas cada m milisegundos (usando -r n -rp m).

`-m:` Para el test y sale de la ejecución cuando 'calls' llamadas son procesadas.

Por ejemplo, estaremos generando con SIPp 7 llamadas cada 2 segundos hacia un *uas* situado en la dirección IP 192.168.1.1 si ejecutamos:

```
# ./sipp -sn uac -r 7 -rp 2000 192.168.1.1
```

Para pausar el tráfico, pulsamos la tecla “p”, y para finalizarlo, la tecla “q”.

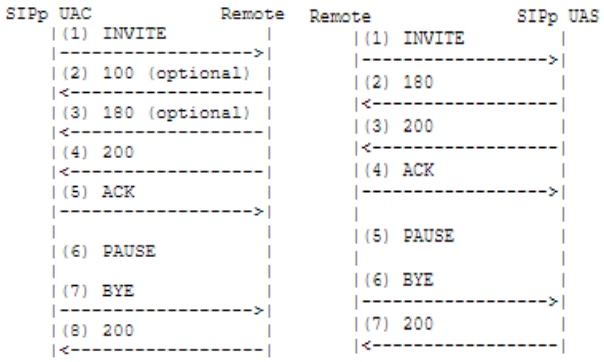
2. **Escenarios personalizados.**

Aunque los escenarios incluidos por defecto son útiles, se puede sacar mucho partido al configurar escenarios personalizados dependiendo de nuestras necesidades.

Los escenarios de SIPp se escriben en *xml* y son áltamente personalizables. Aunque no se ha hecho uso de esta función en el proyecto, se quería nombrar esta interesante posibilidad.

3. **Escenario 1: Carga media-baja.**

Para las primeras pruebas con el Escenario 1, desarrollado en el *Apartado 5.2.1,* se utilizan los escenarios por defecto más sencillos: *uas.xml* y *uac.xml*. Por tanto, el intercambio de mensajes sigue el siguiente esquema:



El código xml de dichos escenarios no ha sido modificado para esta prueba, por lo que se utilizan los códigos básicos de SIPp, que pueden encontrarse en su página web.
Para ejecutarlos, se decide generar una tasa de **4.000** llamadas por segundo, con un límite de 1 millón de llamadas efectuadas (no se alcanzan). Por ello los comandos que se ejecutan son los siguientes:

```
# ./sipp -sn uas
# ./sipp -sn uac -r 4000 -d 1000 -l 10000000 -m 10000000 192.168.66.160
```

4. **Escenario 2 y 3: Carga alta.**

Para las pruebas del segundo y tercer escenario ejecutamos los mismos archivos, pero modificando los parámetros de inicio para realizar la prueba de carga pertinente. Simularemos una tasa de **40.000** llamadas por segundo, con el mismo límite de 1 millón que no se llega alcanzar.

```
# ./sipp -sn uas
# ./sipp -sn uac -r 40000 -d 1000 -l 10000000 -m 10000000 192.168.66.160
```

Bibliografía

[1] M. Boucadair, P. Neves y O. Enarsson, «IP Telephony Interconnection Reference: Challenges, Models, and Engineering,» CRC Press, 2011.

[2] B. Group, «BTS,» 7 Mar 2023. [En línea]. Available: https://www.bts.io/news/bts-wins-meffys-award-for-global-connectivity/. [Último acceso: 20 Enero 2025].

[3] P. Christian, «Ranking the World's Largest International Voice Carriers,» *Teleography*, 2024.

[4] M. Gunturu y R. Aluguri, «Towards Performance Evaluation and Future Applications of eBPF,» 2024.

[5] A. B. Johnston, SIP: understanding the session initiation protocol., Artech House, 2015.

[6] G. N. Purdy, «Linux iptables Pocket Reference: Firewalls, NAT & Accounting.,» O'Reilly Media, Inc, 2004.

[7] J. D. &. H.-J. T. Brouer, «XDP: challenges and future work.,» Proc. Linux Plumbers Conference., 2018.

[8] M. H. N. Mohamed, X. Wang y B. Ravindran, «Understanding the Security of Linux eBPF Subsystem,» *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*, 2023.

[9] D. Soldani, P. Nahi, H. Bour, S. Jafarizadeh, M. Soliman, L. Di Giovanna y F. Risso, «ebpf: A new approach to cloud-native observability, networking and security for current (5g) and future mobile networks (6g and beyond),» *IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS*, 2023.

[10] H. Schulzrinne, «The Session Initiation Protocol (SIP),» Columbia University, New York, NY, USA, 2001.

[11] M. Vieira , M. Castanho, R. Pacífico, E. Santos, E. Junior y L. Vieira, «Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications.,» *ACM Computing Surveys (CSUR), 53(1), 1-36.*, 2020.

[12] R. Oppliger, SSL and TLS: Theory and Practice, Artech House., 2023.

[13] C. T. B. &. W. L. Liu, «Understanding Performance of eBPF Maps.,» *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, 2024.

[14] G. Bertin, «XDP in practice: integrating XDP into our DDoS mitigation pipeline.,» *In Technical Conference on Linux Networking, Netdev (Vol. 2, pp. 1-5). The NetDev Society.*, 2017.

[15] N. Cummings, «How Netflix analyses network traffic at scale using eBPF flow logs,» *figshare*, 2021.

Glosario de términos y siglas.

IP: Internet Protocol.

VoIP: Voice over IP.

RFC: Request for Comments.

SIP: Session Initiation Protocol.

eBPF: Extended Berkeley Packet Filter.

BCC: BFP Compiler Collection.

XDP: eXpress Data Path.

NIC: Network Interface Card.

API: Application Programming Interface.

HTML: HyperText Markup Language.

JSON: JavaScript Object Notation.

RTP: Real-Time Transport Protocol.

TCP: Transmission Control Protocol.

UDP: User Datagram Protocol.

ICMP: Internet Control Message Protocol.

VM: Virtual Machine.