ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA
DIPARTIMENTO DI
INGEGNERIA DELL'ENERGIA ELETTRICA E DELL'INFORMAZIONE
"GUGLIELMO MARCONI"

Corso di laurea in ingegneria
Elettronica per l'energia e l'informazione

Utilizzo di programmi extended Berkeley Packet Filter (eBPF) per implementare funzionalità di rete in Linux

Elaborato in
Laboratorio di Reti e Programmazione di Dispositivi Mobili

Relatore
Prof. Cerroni Walter

Presentata da
Miccoli Francesco

Anno Accademico 2018/2019

Prima di procedere con l'inizio di questo breve elaborato scritto, ci tenevo a porgere un ringraziamento a tutti coloro che mi hanno sempre sostenuto e hanno creduto in me, rimanendo sempre presenti nel momento del bisogno. Occorre cercare di tenersi vicino persone così in caso si abbia la fortuna di trovarle, senza dare mai niente per scontato. In primis un ringraziamento particolare va ai miei genitori ai quali dedico questo lavoro di tesi che segna la fine di un importante percorso.

INDICE

INTRODUZIONE

Negli ultimi tempi il traffico dei dati ha avuto un incremento esponenziale e il numero di dispositivi connessi alla rete supera i 10 miliardi, per questo i requisiti per il 5G sono reti mobili con una maggiore flessibilità ed efficienza energetica, che permettano di aumentare la capacità di trasmissione dati. Per raggiungere questi obiettivi si tende ad orientarsi sempre più verso la tecnologia Software Defined Networking (SDN), la cui architettura consiste nel separare il controllo ed elaborazione dei pacchetti (Control Plane) dall'indirizzamento verso l'hop successivo (Data Plane). Quest'approccio permette la gestione software delle reti tramite NFV (Network Function Virtualization) rendendola più dinamica, consentendole di adattarsi alle esigenze e garantire una maggior sicurezza; inoltre l'implementazione di funzionalità di rete a livello software e non hardware, utilizzando server virtuali, comporta vantaggi economici oltre che facilitare la risoluzione di malfunzionamenti, ridurre gli spazi occupati e permettere una riconfigurazione della rete più agevole.

Per rendere più dinamica la rete occorre poter aumentare la velocità di processamento dei dati. Uno dei metodi che può essere utilizzato consiste nell'implementare un programma *eBPF* di tipo *XDP*, in Linux, il quale permette di accedere ai metadati dei pacchetti in arrivo e agire di conseguenza in base alle informazioni ricavate, ottenendo un esempio di firewall virtualizzato su una macchina opportunamente configurata. Il vantaggio di tale approccio è il poter accedere al contenuto dei pacchetti a livello kernel, permettendone così l'analisi immediata evitando di memorizzare grandi quantità di dati da passare a livello utente. Ciò permette di ridurre l'overhead, analizzando il traffico prima di memorizzarlo, garantendo un aumento di prestazioni.

L'obiettivo del seguente elaborato è di implementare un firewall in *XDP* e mostrarne le prestazioni attraverso test, cercando di migliorare i risultati ottenuti utilizzando le mappe, le quali con la loro aggiunta garantiscono un processamento più veloce dei dati.

Pertanto il seguente lavoro di tesi introduce nel primo capitolo l'*eBPF* parlando della sua struttura e funzionamento, soffermandosi sui programmi di tipo *XDP*; a seguire nel secondo capitolo vengono messi a confronto *XDP* ed *iptables*. Nel capitolo 3 viene spiegato come configurare il kernel per abilitarlo all'utilizzo dei programmi *XDP* e viene argomentato il metodo di caricamento di tali programmi. Nel capitolo 4 vengono esposti vari tipi di mappe e sono mostrati esempi per alcune di esse. Infine nel quinto capitolo sono proposti due programmi aventi lo stesso scopo ma realizzati in modo diverso, e ne sono riportati i test effettuati.

CAPITOLO 1

eBPF & XDP

L'extended Berkeley Packet Filter è la versione estesa del BPF, esistente dal 1992, il cui nome dice poco, infatti oltre al riferimento ad una cittadina della California le uniche parole che danno un significato sono "packet filter" le quali però indicano solamente uno dei tanti utilizzi per il quale l'eBPF può essere sfruttato.

L'eBPF è una macchina virtuale presente nel kernel di Linux, dalla versione 3.18 (Dicembre 2014), che permette di eseguire in maniera sicura un programma in byte-code direttamente nel kernel poiché interpretato dalla macchina virtuale, evitando l'uso di moduli nel kernel. In questo modo è possibile spostare un eventuale filtro di rete dallo spazio utente verso lo spazio kernel così da rendere molto più efficiente il processing dei dati, poiché all'arrivo progressivo dei pacchetti, questi vengono immediatamente analizzati, senza avere il bisogno di copiarli prima nel user-space, ottenendo notevoli guadagni in termini di velocità di elaborazione (fig.1). Bisogna considerare che trovandosi nel kernel i programmi BPF devono sottostare a numerose limitazioni, tra le quali il numero di istruzioni nel codice, per evitare la compromissione dell'intero sistema operativo. Pertanto negli ultimi anni è nata l'esigenza di proporre un'evoluzione del Berkeley Packet Filter, ovvero la sua versione estesa, la quale è così chiamata perché grazie all'introduzione di nuove strutture dati, le Mappe, è possibile eseguire molte più operazioni e soprattutto creare collegamenti tra la zona utente e quella kernel per poter bypassare i limiti precedentemente imposti.

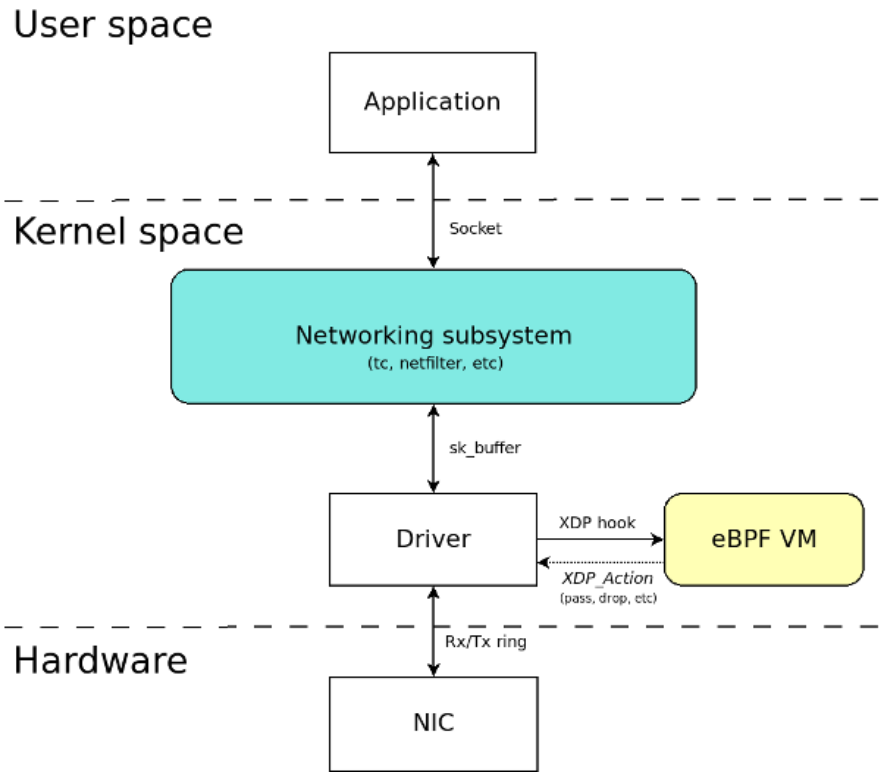

Figura 1. Linux kernel stack con XDP [1].

1.1 Architettura e funzionamento

Per prima cosa occorre fare chiarezza su quella che è la struttura della tecnologia sopra citata e quali componenti ne garantiscono il corretto funzionamento.

Essendo BPF una macchina virtuale, definisce un ambiente in cui vengono applicati i programmi, la quale ha una memoria dove sono presenti le istruzioni da eseguire sul pacchetto in elaborazione, i registri A e X (accumulatore e indice) e un contatore di programmi implicito. In seguito alla sua creazione evidenti cambiamenti nel codice vi sono stati nel 2011, quando Eric Dumazet ha trasformato l'interprete del BPF in un JIT (just in time) in modo da compilare il programma durante l'esecuzione così da velocizzarne il processo, e nel 2014 con Alexei Starovoitov che portò una nuova versione del JIT, il quale utilizza l'architettura su cui si basa oggi l'eBPF.

L'architettura è simile ad un x86-64, infatti l'eBPF utilizza 11 registri a 64 bit (non più solo i registri A e X), e i programmi eBPF hanno uno spazio di

stack limitato a 512 byte, condizione controllata dal verificatore nel kernel, che può essere superata come vedremo nei capitoli 4 e 5, con l'utilizzo delle chiamate a coda per caricare un nuovo programma successivo a quello presente e allungare così il set d'istruzioni di tipo RISC. Le istruzioni vengono passate sottoforma di file C al primo compilatore, LLVM, ovvero *Low Level Virtual Machine*, che è un'infrastruttura di compilazione scritta in C++ che riesce a compilare il codice C in byte-code, linguaggio di livello intermedio, e memorizzarlo nel file oggetto; in seguito quando il file in byte-code viene caricato nel kernel, se rispetta le norme imposte dal verificatore, viene passato al compilatore JIT, il quale traduce le istruzioni dal byte-code nelle istruzioni adatte all'architettura eseguendole allo stesso tempo, "Just In Time", velocizzando l'intero processo.

Di seguito è raffigurato il flusso che segue il programma per essere caricato (fig. 2).

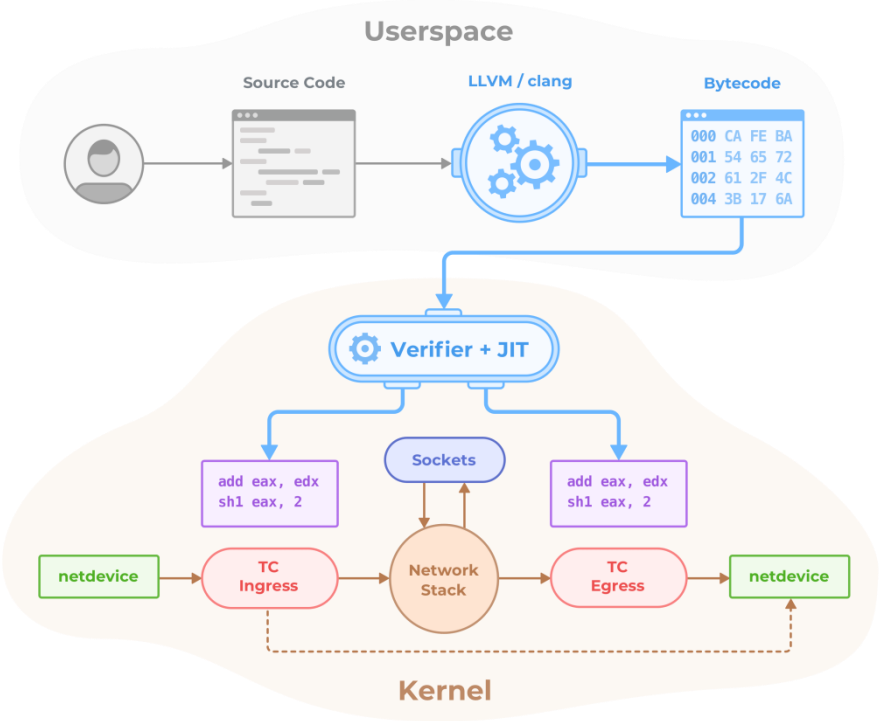

Figura 2. Panoramica eBPF [2].

1.2 Il verificatore

Una volta generato il file oggetto questo è sottoposto alle restrizioni imposte dal verificatore eBPF, infatti non deve contenere loop, deve usare un limitato numero d'istruzioni, 512 byte di stack, e può accedere solamente a spazi di memoria validi, per questo nel file "C" occorre definire tramite puntatori tutte le aree d'intestazione del pacchetto cui si intende accedere con il codice scritto.

```
SEC("xdp_filter")
int _xdp_filter(struct xdp_md *ctx) {       //Funzione a cui è passato
                                            //il puntatore
                                            //alla struttura
                                            //contenente le
                                            //informazioni del
                                            //pacchetto;

void *data_end = (void *)(long)ctx
void *data = (void *)(long)ctx

//Analisi struttura ethernet:

struct ethhdr *eth = data;
if (eth + 1 > data_end) {
        return XDP_ABORTED;
   }
if(ntohs(eth->h_proto) != ETH_P_IP) {
...
   }
```

La parte di codice C scritta sopra mostra un esempio di verifica dei limiti del pacchetto, infatti utilizzando i puntatori "*data*" e "*data_end*" che puntano ad inizio e fine di un'intestazione, in questo caso di quella ethernet, ponendo che se "eth +1" fosse superiore a "data_end" allora il programma restituirebbe "XDP_ABORTED", ovvero si interromperebbe, così è stato possibile accedere ad "h_proto", campo della struttura "struct ethhdr" definita nella libreria *if_ether.h*, che contiene l'ethernet type del pacchetto; una volta conosciuto il tipo di pacchetto il programma si comporterà di conseguenza. Se questo controllo obbligatorio non fosse fatto il verificatore restituirebbe un errore simile a quello sottostante, e il programma non verrebbe caricato nel kernel.

```
Prog section 'prog' rejected: Permission denied (13)!

- Type: 6

- Instructions: 19 (0 over limit)

- License:

Verifier analysis:

0: (61) r1 = *(u32 *)(r1 +0)

1: (71) r2 = *(u8 *)(r1 +13)

invalid access to packet, off=13 size=1, R1(id=0,off=0,r=0)

R1 offset is outside of the packet

Error fetching program/map!
```

1.3 I tipi di programmi eBPF

Oltre al già citato e nel prossimo capitolo approfondito, programma XDP, è importante evidenziare l'esistenza di diversi tipi di programmi eBPF, alcuni dei quali saranno presentati in questo capitolo, e ne saranno riportati esempi nel corso della tesi, tenendo conto che ogni tipo di programma ha le proprie caratteristiche. Di seguito è riportata l'enumerazione dei programmi eBPF presente nella libreria *linux/bpf.h.*

```
 enum bpf_prog_type {
                        BPF_PROG_TYPE_UNSPEC,
                        BPF_PROG_TYPE_SOCKET_FILTER,
                        BPF_PROG_TYPE_KPROBE,
                        BPF_PROG_TYPE_SCHED_CLS,
                        BPF_PROG_TYPE_SCHED_ACT,
                        BPF_PROG_TYPE_TRACEPOINT,
                        BPF_PROG_TYPE_XDP,
                        BPF_PROG_TYPE_PERF_EVENT,
                        BPF_PROG_TYPE_CGROUP_SKB,
                        BPF_PROG_TYPE_CGROUP_SOCK,
                        BPF_PROG_TYPE_LWT_IN,
                        BPF_PROG_TYPE_LWT_OUT,
                        BPF_PROG_TYPE_LWT_XMIT,
                        BPF_PROG_TYPE_SOCK_OPS,
                        BPF_PROG_TYPE_SK_SKB,
                        BPF_PROG_TYPE_CGROUP_DEVICE,
                        BPF_PROG_TYPE_SK_MSG,
                        BPF_PROG_TYPE_RAW_TRACEPOINT,
                        BPF_PROG_TYPE_CGROUP_SOCK_ADDR,
                        BPF_PROG_TYPE_LWT_SEG6LOCAL,
                        BPF_PROG_TYPE_LIRC_MODE2,
                        BPF_PROG_TYPE_SK_REUSEPORT,
```

```
                BPF_PROG_TYPE_FLOW_DISSECTOR,
                BPF_PROG_TYPE_CGROUP_SYSCTL,
                BPF_PROG_TYPE_RAW_TRACEPOINT_WRITABLE,
                BPF_PROG_TYPE_CGROUP_SOCKOPT,
                BPF_PROG_TYPE_TRACING,
        };
```

1.3.1 BPF_PROG_TYPE_SOCKET_FILTER

Questo tipo di programma, di cui sono presenti esempi nel *capitolo Mappe*, appartiene alla categoria "socket", inoltre trattandosi di un filtro appartiene ai primi programmi eBPF nonostante il termine "filtro" sia fuorviante, poiché tale tipo di programma permette la creazione di una *raw socket* a cui ci si collega, e tramite la quale è possibile analizzare il traffico scelto passante per quella socket. I dati che vengono osservati riguardanti il pacchetto non sono altro che metadati, e non è possibile usare funzioni che permettano lo scarto di pacchetti impedendogli il raggiungimento dello spazio utente, poiché con la raw socket possiamo solamente osservare il flusso dati. L'interessante pratica aggiunta con l'introduzione delle mappe è la capacità di poter condividere le statistiche del flusso dati tra il livello applicativo e quello kernel, avendo la possibilità di memorizzare il numero di byte, di pacchetti e restituire l'informazione del tipo di pacchetti (TCP, UDP, ICMP, ecc.) passanti per la raw socket. Di seguito è riportato un esempio utilizzato anche nei capitoli successivi da cui è possibile osservare le funzioni utilizzate per caricare il programma e attaccarlo a una socket.

```
sock = open_raw_sock("ens1f1");   //Apre socket su interfaccia indicata;

assert(setsockopt(sock, SOL_SOCKET   //imposta opzione socket per
, SO_ATTACH_BPF, prog_fd,            //far funzionare il programma eBPF;
sizeof(prog_fd[0]);
```

Viene mostrato come utilizzando la funzione "open_raw_sock(nome interfaccia)" sia possibile aprire una socket sull'interfaccia desiderata, mentre in seguito la funzione "setsockopt()", contenuta nella libreria "*sys/socket.h"* permette, per com'è impostata, di caricare il programma eBPF a livello socket (SOL_SOCKET) passando il file descrittore, mentre la funzione *assert()* permette di continuare l'esecuzione del file solo se la funzione che ha per argomento viene eseguita con successo.

1.3.2 I tipi Kprobe, Tracepoint e Perf Event

Questi tipi di programmi vengono utilizzati per poter fornire un servizio di debugging e tracciamento del traffico dati, come il tcpdump. Vi sono due approcci possibili per poter far scambiare informazioni tra user e kernel space, il primo è quello già citato che prevede l'utilizzo delle mappe, le quali consentono la creazione di array condivisi tra i due spazi, l'altro consiste nell'utilizzare i *perf event buffers*, molto più veloci perché sono memorie con CPU dedicata, che permettono di stampare a video le informazioni che durante l'esecuzione del programma vengono accumulate nei buffer. Uno dei metodi più semplici per effettuare il debug del codice, oppure per condurre un'operazione di tracciamento, è quello di utilizzare una funzione d'aiuto (helper function), che verranno presentate in maniera più approfondita nei capitoli successivi, tranne quella a seguire, poiché utile per l'argomento affrontato in questo capitolo.

```
//Macro per la stampa;
#define bpf_printk(fmt, ...)                    \
({                                  \
           char ____fmt[] = fmt;                \

           bpf_trace_printk(____fmt, sizeof(____fmt),   \
                ##__VA_ARGS__);         \
})
...
bpf_printk("source ip address is %u\n", ip_src);
```

L'esempio mostra come definire la macro della funzione "*printk()"*; richiamandola in seguito consente di stampare i messaggi dal programma eBPF eseguito nel kernel, in questo caso il valore contenuto nella variabile *ip_src*. Normalmente non sarebbe possibile fare molte stampe poiché il numero d'informazioni dovrebbe essere limitato nel lato kernel , ma in questo modo i messaggi vengono salvati nei trace buffer, il cui contenuto può essere visualizzato tramite il seguente comando da terminale:

```
$ sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Per quanto riguarda i programmi di tracciamento veri e propri il loro utilizzo risulta più complesso, di questo tipo ne fanno parte i Kprobe, Perf_Event e Tracepoint (fig. 3), i quali possono essere agganciati ad altri programmi e si attivano quando si verificano specifici eventi all'interno del kernel; di *Tracepoint* ne è trattato un esempio nel quarto capitolo, poiché prima occorre parlare più approfonditamente di programmi user, programmi kern e delle mappe.

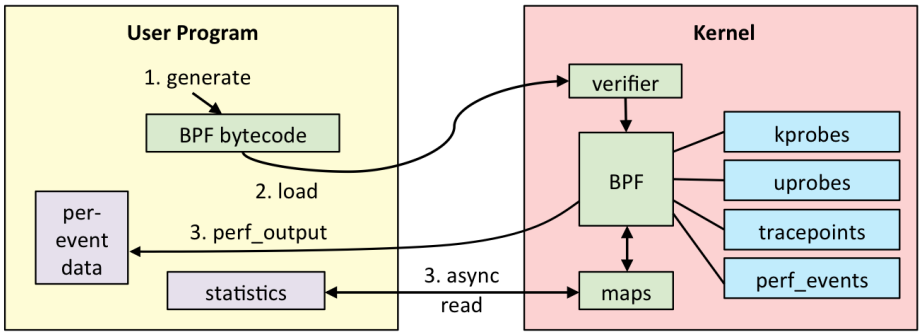

Figura 3. Programmi di tracciamento [3].

1.3.3 XDP

L'XDP o *Express Data Path* è il programma eBPF sul quale è maggiormente incentrata questa tesi, esso è nato come soluzione agli attacchi DDoS poiché occorreva trovare un modo per gestire un traffico molto elevato di dati, dato che *iptables* non era abbastanza veloce. Poiché i programmi eBPF possono essere agganciati in diversi punti nel kernel, *XDP* sfrutta tale caratterista per essere attaccato al driver della scheda di rete, quindi anteposto al network stack, così da permettere l'analisi del traffico dati prima che venga allocata della memoria per memorizzare i pacchetti nei socket buffer (fig.4), tale selezione di dati prima che vengano immagazzinati permette di diminuire l'overhead. Questa tecnologia si contrappone al *Data Plane Developmet Kit* (DPDK), il cui compito è quello di spostare il controllo del traffico di rete dal kernel verso lo user-space, prevedendo dunque una memorizzazione del traffico precedente all'analisi di ciò che viene catturato e per questo anche chiamato *kernel bypass framework*. Ovviamente l'elaborazione viene rallentata, però, essendo tutto eseguito nel spazio utente risulta la mancanza di vincoli per XDP, il quale è

sottoposto al verificatore. Proprio questi limiti sarà obiettivo di questo elaborato scritto superarli, poiché utile poter eseguire un programma a livello kernel mantenendo i vantaggi dello spazio utente.

XDP utilizza la tecnica di *hooking*, ovvero ogni volta che si verifica un evento, per il nostro tipo di programma che arrivi un pacchetto alla scheda di rete, questo viene agganciato e si accede ai suoi metadati con i quali si lavora. Infatti come visto quando trattato il verificatore, il programma xdp in *C* è definito come una funzione che accetta come argomento un puntatore alla struttura contenente i metadati, così definita nella libreria *linux/bpf.h*:

```
struct xdp_md
{
        __u32 data;
        __u32 data_end;
        __u32 data_meta;
        __u32 ingress_ifindex;
        __u32 rx_queue_index;
};
```

Medesimamente per le azioni che può restituire un programma XDP, sono definite nella stessa libreria, e vengono usate per determinare la sorte dei pacchetti, potendo in questo modo dare al programma diverse funzionalità come quella di firewall, NAT oppure di tunneling, dato che una volta ottenuto l'accesso alle varie intestazioni è possibile modificarne i valori, come ad esempio IP sorgente e destinazione, e consentire il passaggio oppure l'interruzione del traffico (fig.4).

```
enum xdp_action {
        XDP_ABORTED = 0, //Programma interrotto;
        XDP_DROP,        //Pacchetto viene scartato;
        XDP_PASS,        //Pacchetto passa;
        XDP_TX,          //Reindirizza il pacchetto
                         //all'interfaccia sorgente;
        XDP_REDIRECT,    //Reindirizza il pacchetto ad
                         //un'altra interfaccia;
};
```

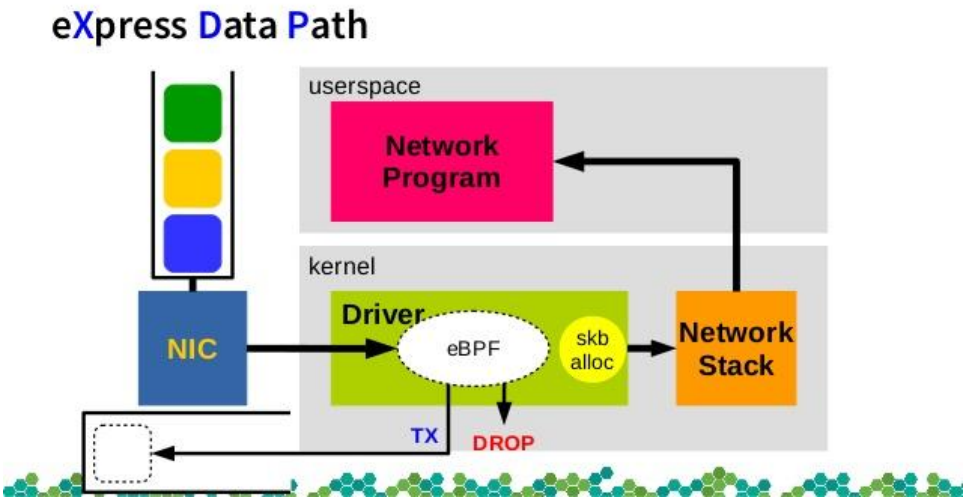

Figura 4. Panoramica XDP [4].

CAPITOLO 2

XDP vs IPTABLES

XDP nato per la funzionalità di scartare i pacchetti indesiderati, ovvero come soluzioni agli attacchi DDoS, potendo anche leggere, scrivere e indirizzare i pacchetti ora questo tipo di programma può coprire un ampio campo di applicazioni pratiche, quindi usato per implementare qualsiasi funzione di rete. Nel seguente capitolo si parlerà del suo utilizzo come firewall messo a confronto con *iptables*, uno strumento integrato nel kernel di linux.

2.1 Iptables

Iptables è largamente utilizzato per configurare il kernel di Linux come firewall e NAT. La sua importanza è dovuta alla possibilità di creare facilmente un filtro utilizzando regole di filtraggio impilate secondo una struttura tabellare, lette dall'alto verso il basso, tenendo conto che le regole scritte prima hanno priorità su quelle a seguire, ed inoltre garantisce la possibilità di mantenere una connessione di stato, potendo ad esempio evitare il controllo inutile di un flusso di pacchetti provenienti da una sorgente la cui comunicazione è già stata controllata e stabilita in precedenza, come nel caso mostrato:

`iptables -P FORWARD DROP`	--->	Applico policy di default deny.
`iptables -A FORWARD -i eth0 -m state --state NEW -j ACCEPT`	--->	Accetto tutto in ingresso all' interfaccia eth0 di stato NEW.
`iptables -A FORWARD -i ppp0 -d 87.15.12.0/24 -p tcp --dport 80 -m state --state NEW -j ACCEPT`	--->	Accetto tutto in ingresso all'interfaccia ppp0 , di tipo TCP diretto alla porta 80 con stato NEW, della rete 87.15.12.0/24.
`iptables -I FORWARD 1 -m state --state`	--->	Inserisco regola

`ESTABLISHED -j ACCEPT`	`in prima posizione; accetto tutti i pacchetti il cui flusso è già stato accettato in precedenza`

Il problema nell'utilizzo di *iptables* è che per intercettare il traffico di pacchetti utilizza *netfilter*, un software il cui funzionamento consiste nel far attraversare il network stack del kernel prima di permettere il processamento dei pacchetti, "perdendo tempo". Arrivati a questo punto si presenta un altro problema di *iptables* ovvero quello di utilizzare una ricerca sequenziale nella tabella per trovare eventuali corrispondenze, rendendo l'intero procedimento davvero lento se l'intenzione è quella di implementare un firewall con molte istruzioni e la regola ricercata è una delle ultime, considerando inoltre che per ogni pacchetto che passa occorre verificare la lista di regole. Ciò ha fatto in modo che si cercasse una soluzione a questi problemi, anche perché rispetto ad una ventina di anni fa, quando nasceva *iptables*, che venivano utilizzate una dozzina di regole, oggi le tabelle possono essere composte anche da migliaia di istruzioni, e verificare ognuna di esse è diventato improponibile, poiché a ciò ne consegue una latenza troppo elevata. Anche con l'introduzione di *ipset*, un framework nel kernel di Linux, che permette di comprimere il numero di regole iptables andando a memorizzare quelle che riguardano i numeri IP e di porta in una tabella hash. Per questo importanti compagnie come Facebook, Google e Netflix hanno deciso di spostarsi verso l'utilizzo di nuove tecnologie, come BPF, per quanto riguarda l'ambito della sicurezza di rete, il load balancing e il monitoraggio dei dati. Infatti tutto ciò che può essere fatto con iptables si può fare con un programma XDP, inoltre con discreti vantaggi, come mostra la figura 5, dove è riportato un confronto di prestazioni di uno stesso firewall realizzato con tecnologie differenti. La figura, presentata al FRnOG 30[1] da parte di Quentin Monnet, contiene

[1] Gruppo che riunisce persone interessate al settore della sicurezza informatica in Francia.

anche le prestazioni ottenute con *nftables*, la versione più aggiornata di *iptables*, con *throughput*[2] migliore rispetto a quest'ultimo.

Nonostante la dimostrazione di tecnologie migliori, infatti con un programma XDP si può raggiungere una bit-rate quattro volte superiore, *iptables* ancora oggi viene utilizzato e non è del tutto scomparso, poiché molto diffuso, e ciò ne garantisce un continuo utilizzo per almeno un'altra decina di anni.

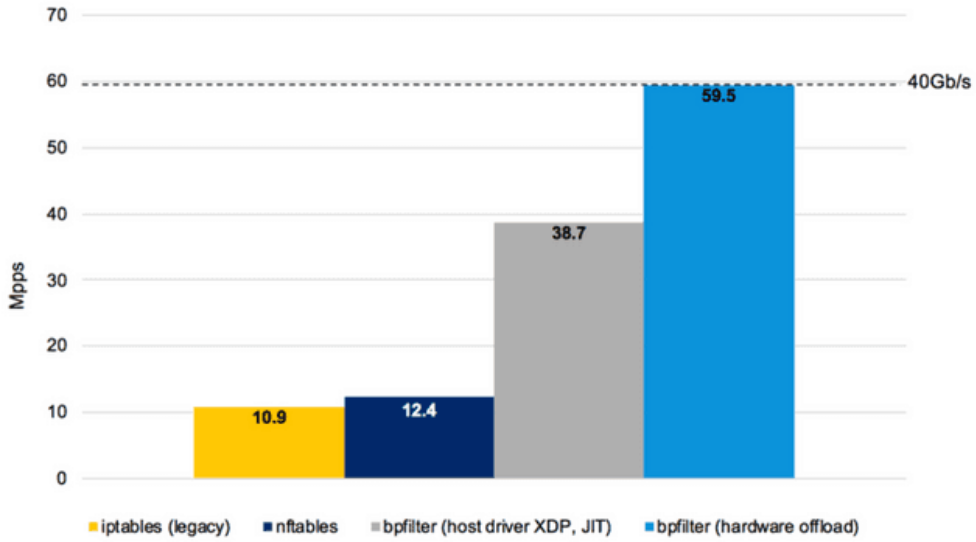

Figura 5. Tecnologie per firewall a confronto[5]

[2] Capacità effettiva di un canale, misurata in Mpps(Maximum packets per second), 1Gbps Ethernet richiede 1,488Mpps.

CAPITOLO 3

CONFIGURAZIONE DEL KERNEL

Prima di proporre codici di programmi eBPF occorre illustrare come costruire il kernel affinché supporti XDP e ci consenta di poter accedere a tutte le librerie che saranno necessarie per lo sviluppo dei codici, com'è capitato di incontrare nei capitoli precedenti, incluse le *helper functions*, che consentono di ampliare le funzionalità dei programmi attraverso l'utilizzo di chiamata a sistema, *system call*.

Per prima cosa occorre scaricare l'albero, ovvero l'insieme dei file sorgenti relativi ad una determinata versione del kernel Linux che si desidera utilizzare, strutturati in uno schema ad albero, nel caso di questo elaborato i codici sviluppati sono stati creati utilizzando l'insieme dei file corrispondente a questo link:

```
https://github.com/torvalds/linux/tree/v5.4
```

Pertanto è stato scaricato utilizzando il seguente comando:

```
$ git clone https://github.com/torvalds/linux.git

$ cd linux
```

Ora bisogna scegliere la versione del kernel da compilare:

```
$ git checkout v5.4 -b v5.4
```

Il prossimo passaggio consiste nel configurare il tutto, per far ciò serve un file *.config*, il metodo più semplice per ricavarselo è quello di copiarne uno già esistente dal kernel di sistema, ed è possibile farlo nel seguente modo:

```
$ uname -r

5.4.0-050400-generic

$ cp /boot/config-5.4.0-050400-generic ./.config
```

In seguito si può utilizzare la configurazione di default per costruire l'albero:

```
$ make defconfig
```

Infine per installare il kernel e poter utilizzare i file d'intestazione occorre eseguire gli ultimi due comandi per completare il processo:

```
$ sudo make install

$ make headers_install
```

Ora che il kernel è pronto bisogna assicurarsi che siano presenti tutti gli strumenti necessari, come ad esempio il compilatore LLVM, pertanto per quanto riguarda la distribuzione Ubuntu bisogna installare l'occorrente in questo modo:

```
$ sudo apt install clang llvm libelf-dev gcc-multilib

$ sudo apt install linux-tools-$(uname -r)
```

In quest'albero, al percorso "*sample/bpf",* sono presenti diversi esempi di programmi eBPF, i quali sfruttando il *Makefile*, presente all'interno della suddetta cartella, è possibile compilarli tutti in una volta:

```
$ sudo make samples/bpf/
```

generando così i file eseguibili che ne permettono il caricamento su un'interfaccia.

3.1 Il caricamento usando il file "_user.c"

Come si può notare gli esempi compilati, sono suddivisi in due file, i quali terminano con "_kern.c" e "_user.c", questo perché facendo uso di *system calls* è possibile usare il file "user" per caricare il programma eBPF scritto nel file "_kern.c", questo metodo è fondamentale se occorre condividere le informazioni tra livello utente e kernel tramite l'utilizzo delle mappe, com'è possibile vedere nel capitolo successivo, infatti se si utilizzasse un altro metodo, come *iproute2*, che prevede l'impiego di un unico file, non si riuscirebbero a condividere le informazioni.

Oltre all'albero scaricato per eseguire i test presenti in questo elaborato, sono stati scaricati altri esempi di programmi con il seguente comando:

```
$ git clone --recurse-submodules https://github.com/xdp-
project/xdp-tutorial
```

Gli esempi presenti in *xdp-tutorial* possono essere compilati con il comando "*make*" e differiscono dagli esempi precedenti poiché utilizzano le librerie *libbpf* per attaccare nel kernel Linux i programmi "_kern.o", generati dal compilatore LLVM; mentre i codici in "*sample/bpf"* per essere caricati fanno affidamento al file *bpf_load.o* all'interno della stessa cartella dove sono presenti i programmi.

3.2 Esempio di programma XDP

Ora si dispone di due cartelle di lavoro, le quali contengono programmi che vengono caricati con due metodi differenti, nel corso dei successivi due capitoli verranno utilizzati entrambi i metodi, poiché in base allo scopo che ha il programma da implementare risulta più semplice usare un certo tipo di metodo rispetto all'altro.

È proposto un semplice programma xdp il cui compito è di bloccare tutto il traffico scartando i pacchetti che arrivano all'interfaccia di rete. Per far ciò nell'esempio è utilizzato l'approccio di xdp-tutorial.

Per prima cosa si accede alla cartella *xdp-tutorial* e al suo interno si crea la cartella che conterrà il nostro programma. Una volta fatto ciò vi si accede e si creano i file "user" e "kern":

```
root@hp187:~/linux/xdp-tutorial/xdp_drop# ll
-rw-r--r--  1 root root    207 Dec  4 15:59 xdp_drop_kern.c
-rw-r--r--  1 root root   4860 Dec  4 15:59 xdp_drop_user.c
```

I quali avranno i seguenti codici, *xdp_drop_kern.c* :

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>
SEC("xdp_drop")
int  xdp_prog_simple(struct xdp_md *ctx)
{
        return XDP_DROP;
}
char _license[] SEC("license") = "GPL";
```

Il codice riportato è ciò che il nostro programma di tipo XDP andrà a fare, ovvero accederà ai pacchetti in arrivo e li scarta, dato che l'unico comando

della funzione è di restituire *XDP_DROP*. Particolarità comune a tutti i programmi è la necessità di una licenza per funzionare, e inoltre la funzione principale deve essere introdotta da *SEC("nome_sezione")* la quale definisce una sezione del file dedicata al programma eBPF in questione; è da rilevare che all'interno di un file possono essere presenti più sezioni e quindi più programmi eBPF, tra i quali sarà caricato sull'interfaccia solo quello scelto.

xdp_drop_user.c (codice completo riportato nell'appendice **"A"**) :

```
…
//Assegna al puntatore "filename" il nome del file di default.
static const char *default_filename = "xdp_drop_kern.o";

//Assegna il programma di default, contenuto nel file, a "prog_sec".
static const char *default_progsec = "xdp_drop";
…
// int argc: contiene il numero di stringhe inserite dall'utente a linea
// di comando;
// char *argv[]: l'array che contiene le stringhe inserite dall'utente a
// linea di comando,
// (ogni elemento dell'array è un puntatore a carattere).
int main(int argc, char **argv)
{
  struct bpf_object *bpf_obj;
  struct config cfg = {
  .xdp_flags = XDP_FLAGS_UPDATE_IF_NOEXIST | XDP_FLAGS_DRV_MODE,
  .ifindex   = -1, //Inizialmente nessuna interfaccia impostata;
  .do_unload = false,
  };
//Copia il nome del file oggetto nel campo della struttura config.
strncpy(cfg.filename, default_filename, sizeof(cfg.filename));

//Copia il nome del programma
strncpy(cfg.progsec,  default_progsec,  sizeof(cfg.progsec));
…
bpf_obj = __load_bpf_and_xdp_attach(&cfg);
…
return EXIT_OK;
}
```

Sono state riportate alcune delle parti più importanti del codice che permettono il caricamento del file *xdp_drop_kern.o*. Il file viene caricato assegnando a due puntatori il nome del file *"_kern.o"* e il nome della sezione, che si desidera caricare.

In seguito nel "*main"* del programma è richiamata la funzione "*__load_bpf_and_xdp_attach(&cfg)*", la quale accetta come argomento il puntatore alla struttura "*config*" definita nella libreria "*common_defines.h*", e avente come campi le scelte per il caricamento del programma inserite da terminale al momento dell'avvio. Tale funzione utilizza a sua volta quattro helper functions:

```
bpf_prog_load_xattr(&prog_load_attr, &obj, &first_prog_fd);

bpf_prog = bpf_object__find_program_by_title(bpf_obj, cfg->progsec);

prog_fd = bpf_program__fd(bpf_prog);

xdp_link_attach(cfg->ifindex, cfg->xdp_flags, prog_fd);
```

La prima carica il file oggetto nel kernel via syscall, la seconda permette di verificare se la sezione impostata è presente nel file caricato, se il risultato è affermativo, allora la terza funzione ricava il file descrittore, indispensabile per caricare definitivamente il programma con la quarta funzione che accetta come argomento anche l'interfaccia su cui va caricato.

3.2.1 Compilazione e avvio del programma

Per compilare i programmi sopra riportati e generare l'eseguibile occorre creare un file, chiamarlo *Makefile* e inserire al suo interno il seguente codice:

```
# SPDX-License-Identifier: (GPL-2.0 OR BSD-2-Clause)

# Departing from the implicit _user.c scheme

XDP_TARGETS  := xdp_drop_kern

USER_TARGETS := xdp_drop_user

LIBBPF_DIR = ../libbpf/src/

COMMON_DIR = ../common/

include $(COMMON_DIR)/common.mk
```

Infine lanciando il comando *make* da terminale è possibile compilare il tutto e ottenere il seguente risultato:

```
root@hp187:~/linux/xdp-tutorial/xdp_drop# ll
```

```
-rw-r--r--  1 root root    281 Dec  4 15:59 Makefile
-rw-r--r--  1 root root    207 Dec  4 15:59 xdp_drop_kern.c
-rw-r--r--  1 root root   4555 Dec  4 15:59 xdp_drop_kern.ll
-rw-r--r--  1 root root   3928 Dec  4 15:59 xdp_drop_kern.o
-rwxr-xr-x  1 root root 568776 Dec  4 15:59 xdp_drop_user*
-rw-r--r--  1 root root   4860 Dec  4 15:59 xdp_drop_user.c
```

Ora per caricare il programma ad esempio sull'interfaccia *ens1f1* occorre eseguire da command line:

```
root@hp187:~/linux/xdp-tutorial/xdp_drop#./xdp_drop_user --dev
ens1f1 --force
```

Grazie all'opzione "force" è possibile caricare un programma xdp anche se sull'interfaccia ne è presente già un altro, poiché provvede alla sua automatica sostituzione perché su un'interfaccia può essere caricato un solo programma alla volta. Nel caso si voglia togliere il programma xdp senza sostituirlo con un altro occorre cambiare il comando sostituendo "force" con "unload". Ciò rende il caricamento e la gestione dei programmi veloce con il metodo proposto da *xdp-tutorial*.

Altrimenti in caso di necessità lanciando il seguente comando è sempre possibile rimuovere il programma caricato.

```
ip link set dev ens1f1 xdp_drop off
```

CAPITOLO 4

LE MAPPE

Da com'è stato presentato, eBPF può essere adoperato come soluzione ai problemi di networking, dovuti a una quantità ragguardevole di dati, che devono essere indirizzati ed elaborati all'interno della rete e nei nostri dispositivi il più velocemente possibile. Unico svantaggio sono le limitazioni imposte su questo tipo di tecnologia, pertanto è normale la presenza di un *trade off*, infatti prendendo come esempio l'implementazione di un firewall in XDP, questo analizza il traffico più velocemente rispetto ad altre tecnologie ma il controllo che può compiere sarà meno meticoloso avendo a disposizione un numero più ristretto d'istruzioni. Questo limite può essere superato tramite l'utilizzo delle mappe, ovvero archivi che immagazzinano i dati come coppia chiave/valore, dove ogni chiave è da considerarsi un identificatore univoco a un determinato valore. I valori contenuti nelle mappe sono condivisi da livello utente a kernel consentendo di evitare il sovraccarico d'istruzioni nell'area sottoposta al controllo del verificatore, poiché la mappa è uno spazio di memoria che è allocata in *user-space* dal programma eBPF, nel quale devono essere definite le mappe da utilizzare.

L'elenco delle mappe disponibili è contenuto nell'enumerazione "*bpf_map_type"* presente nella libreria "*linux/bpf.h"* dell'albero scaricato nel capitolo 3.

```
enum bpf_map_type {
                        BPF_MAP_TYPE_UNSPEC,
                        BPF_MAP_TYPE_HASH,
                        BPF_MAP_TYPE_ARRAY,
                        BPF_MAP_TYPE_PROG_ARRAY,
                        BPF_MAP_TYPE_PERF_EVENT_ARRAY,
                        BPF_MAP_TYPE_PERCPU_HASH,
                        BPF_MAP_TYPE_PERCPU_ARRAY,
                        BPF_MAP_TYPE_STACK_TRACE,
                        BPF_MAP_TYPE_CGROUP_ARRAY,
                        BPF_MAP_TYPE_LRU_HASH,
                        BPF_MAP_TYPE_LRU_PERCPU_HASH,
                        BPF_MAP_TYPE_LPM_TRIE,
```

```
                    BPF_MAP_TYPE_ARRAY_OF_MAPS,
                    BPF_MAP_TYPE_HASH_OF_MAPS,
                    BPF_MAP_TYPE_DEVMAP,
                    BPF_MAP_TYPE_SOCKMAP,
                    BPF_MAP_TYPE_CPUMAP,
                    BPF_MAP_TYPE_XSKMAP,
                    BPF_MAP_TYPE_SOCKHASH,
                    BPF_MAP_TYPE_CGROUP_STORAGE,
                    BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
                };
```

La prima cosa che occorre fare per utilizzare una qualunque delle mappe riportate è quella di definirla nel file "_kern.c" tramite la struttura globale "*bpf_map_def"* con la sezione SEC("maps"):

```
struct bpf_map_def SEC("maps") hello_map = {
  .type          = BPF_MAP_TYPE_ARRAY,
  .key_size      = sizeof(__u32),
  .value_size    = sizeof(__u64),
  .max_entries   = 10,
};
```

Come riporta l'esempio per definire una mappa correttamente occorre assegnare certi valori ai campi della struttura, come il tipo di mappa che si intende creare, le dimensioni delle chiavi e dei corrispondenti valori, ed infine il numero massimo di ingressi accettati. Soprattutto la sezione per com'è definita rappresenta una mappa chiamata "*hello_map",* di tipo *ARRAY,* con dieci valori rappresentati con *64 bit unsigned*, le cui chiavi hanno dimensione di un *intero senza segno.*

4.1 Helper functions per le mappe

Una volta spiegato come si crea una mappa, è importante sapere come poterne averne accesso sia da user-space sia da kernel-space. Questo è possibile tramite chiamate a sistema, in particolare le funzioni BPF che lo permettono vengono illustrate qui di seguito.

La funzione "*_update_*" permette di memorizzare il valore, indicato tra i suoi argomenti, in una determinata mappa facendolo corrispondere a una chiave, specificata sempre tra gli argomenti. Se l'operazione ha successo, restituisce "0", altrimenti un valore negativo:

```
int bpf_map_update_elem(struct bpf_map *map, const void *key,
```

```
                                const void *value, u64 flags)
```

inoltre in base al flag (tabella 1) utilizzato, s'informa il sistema come si deve comportare con la chiave fornita, e quindi se il valore da memorizzare è nuovo oppure da sovrascrivere.

FLAGS:	
BPF_NOEXIST	---> La chiave non deve già esistere;
BPF_EXIST	---> La chiave deve già esistere;
BPF_ANY	---> Non pone condizione sull'esistenza della chiave;

Tabella 1. Flag disponibili.

Invece con "*_lookup_"* è possibile cercare il valore in una mappa, della quale è mostrato il nome tra gli argomenti, corrispondente alla chiave fornita. Tale funzione restituisce il valore cercato oppure *"NULL"* se non è trovato.

```
void *bpf_map_lookup_elem(struct bpf_map *map, const void *key)
```

Un'altra funzionalità che può tornare utile è avere la possibilità di scorrere le chiavi di una mappa, ciò è possibile grazie alla seguente funzione:

```
int bpf_map_get_next_key(int fd, const void *key,
                         void *next_key)
```

Se è trovata una chiave successiva, viene restituito zero, altrimenti "-1" ed "*errno"* viene impostato con il flag "*ENOENT"* in caso le chiavi siano finite.

Infine un'ultima chiamata di cui ci si può servire è *"_delete_"* che semplicemente cancella il valore associato alla chiave proposta, e restituisce "0" se il procedimento va a buon fine, in caso contrario ritorna un valore negativo.

```
int bpf_map_delete_elem(struct bpf_map *map, const void *key)
```

4.2 Tipi di mappe

Come anticipato ad inizio del quarto capitolo esistono diversi tipi di mappe, ognuna delle quali specializzata per certe funzionalità e avente specifiche proprie. Il seguente paragrafo è suddiviso in sottocapitoli, che trattano alcuni tipi di mappe e ne riportano degli esempi nelle appendici "B" e "C".

4.2.1 BPF_MAP_TYPE_ARRAY & BPF_MAP_TYPE_HASH

Queste due mappe fanno parte delle prime, introdotte dalla versione 3.19 del kernel di Linux. Entrambe allocano uno spazio di memoria equivalente ad un array, avente dimensione pari al numero d'ingressi massimi impostati nella definizione della mappa. Invece per quanto riguarda le differenze, con la mappa "_TYPE_ARRAY" le chiavi sono vincolate ad essere di 4 byte, i valori che bisogna attribuire alle chiavi devono essere identici a quelli degli indici di un array, compresi tra "0" e "*max_entries-1".* In aggiunta sempre per lo stesso tipo di mappa, come vale per la "*BPF_MAP_TYPE_PERCPU_ARRAY*", se utilizzata la funzione *"_update_"* non può essere usato il flag "*BPF_NOEXIST"*, poiché le mappe array inizializzano tutti i valori a zero al momento della creazione della mappa, tutti gli elementi esistono già, pertanto se inserito tale flag viene restituito un errore.

4.2.2 BPF_MAP_TYPE_PERCPU_

Sia la "*_PERCPU_ARRAY"* che la "*_PERCPU_HASH"* sono rese disponibili da Maggio 2016, con la versione 4.6 del kernel. Queste mappe presentano le stesse caratteristiche delle loro versioni precedenti, l'unica eccezione è che ne viene creata una per ogni core della CPU, in modo che processori diversi non confondano i dati tra di loro, cosi da aumentare le performance.

4.2.3 BPF_MAP_TYPE_PROG_ARRAY

Mappa aggiunta a partire della versione 4.2 del kernel; consiste in un array avente per valori i descrittori dei programmi eBPF presenti nel file oggetto.

La dimensione delle chiavi e dei rispettivi valori devono essere di 4 byte, inoltre questa struttura dati è utilizzata con la funzione:

```
int bpf_tail_call(void *ctx, struct bpf_map *prog_array_map,
                  u32 index)
```

La funzione riportata permette di gestire le chiamate a coda, ovvero accettando come argomenti il puntatore ai metadati, il puntatore alla mappa contenente i descrittori, e l'indice corrispondente al file descrittore di uno dei programmi definiti nel file "_kern.c", è possibile saltare a quel programma. In questo modo è possibile eseguire più programmi presenti nel file oggetto, allungando così il numero di istruzioni consentite, oppure decidere di far eseguire un programma in base al tipo di pacchetto ricevuto. Sono però da tener presente alcune condizioni da rispettare, infatti al momento il numero massimo di programmi da poter essere chiamati in successione è 32, richiedono essere dello stesso tipo e una volta lasciata l'esecuzione di un programma non vi si può ritornare.

CAPITOLO 5

IMPLEMENTAZIONE DI UN SEMPLICE FIREWALL CON XDP

In questo quinto ed ultimo capitolo dell'elaborato viene fatto un confronto tra due programmi che implementano lo stesso firewall, avendo due codici dissomiglianti, dato che uno usa le mappe mentre l'altro no. L'obiettivo è quello di verificare la presenza di differenti prestazioni, e se con le mappe sia possibile superare l'obbligatorietà di istruzioni a cui i programmi eBPF sono sottoposti. Per questi test i programmi sono stati eseguiti andando ad incrementare ogni volta il numero di istruzioni.

Il firewall proposto in due diverse versioni, sempre xdp, permette il passaggio di un traffico che utilizza un numero di porta presente nella *white list*, escluso il traffico della porta 9999. Di seguito vengono argomentati i file: "*firewall_N_kern.c"* e "*firewall_N_noMappe_kern.c"*[3] con i rispettivi "_user.c".

5.1 Senza mappe

Il programma che non fa uso di mappe è stato creato all'interno della cartella "*xdp-tutorial",* come l'esempio del capitolo 3, pertanto utilizza le librerie libbpf, le quali hanno già definito in *"parsing_helpers.h"* le funzioni che vanno ad analizzare i limiti delle intestazioni a cui si vuole accedere, e quindi non presenti nel codice perché sono richiamate.

Per com' è definita la sezione all'interno del file lato kernel, alla venuta di un pacchetto si accede all'intestazione IP, e vengono salvati in due variabili l'IP sorgente e l'IP destinazione; questo serve per poter imporre la condizione di analizzare solo il flusso dati che ha una certa destinazione e una certa sorgente, lasciando passare tutto il resto. Da notare che quando si scambiano informazioni tra sistema e pacchetti di rete, occorre utilizzare

[3] "N" sta per numero di istruzioni.

funzioni che adattino il formato, da variabili in memoria a pacchetto che deve uscire e viceversa[4], permettendo di rispettare le convenzioni "little endian" e "big endian". Infine solo se il pacchetto è di tipo TCP oppure UDP il programma passa ad analizzare le possibili porte, bloccando il traffico solo se individuata la 9999. Per aumentare il numero di istruzioni occorre accrescere il numero di porte da analizzare tramite comandi "if". I codici sono riportati nell'appendice D.

5.2 Con mappe

Nel caso con le mappe vale ciò riportato nella sottosezione precedente, con l'unica differenza che i file vengono creati nella cartella "*samples/bpf"* e quindi occorre far riferimento a diversi metodi di caricamento e di configurazione del Makefile per poter far funzionare il nuovo programma da creare. Vengono riportati di seguito le parti del file da aggiungere per compilare i file "*firewall_10_kern.c"* e "*firewall_10_user.c"*:

```
...
hostprogs-y += firewall_10
...
firewall_10-objs := bpf_load.o firewall_10_user.o
...
always += firewall_10_kern.o
...
```

Come si può verificare nell'appendice E, nel file lato kernel sono definite una mappa hash, alla quale sono assegnate come chiavi i numeri delle porte e come corrispondenti valori "XDP_PASS"[5] o "XDP_DROP"[6], e un'altra mappa di tipo array per cpu, la quale viene utilizzata per contare eventuali pacchetti scartati in caso di utilizzo della porta 9999. Del resto dalla zona kernel viene solo fatto un lookup per vedere se una delle chiavi corrisponde alla porta del pacchetto in arrivo, mentre le chiavi e i correlati valori vengono assegnati con un ciclo "for" nello spazio utente, dove il limite di memoria è quello del disco ed i loop sono concessi.

[4] Funzioni "_htonl" e "_ntohl".
[5] XDP_PASS = 2.
[6] XDP_DROP = 1.

A differenza del programma senza mappe, per aggiungere numeri di porta da confrontare, e quindi istruzioni, basta cambiare la dimensione della mappa hash e il numero di iterazioni del ciclo “for” nel programma utente.

5.3 Test

Per effettuare i test si è fatto uso di tre server messi a disposizione da CloudLab, un ambiente che consente di utilizzare architetture su cui effettuare prove in ambito di ricerca. In particolare per i test sono stati utilizzati tre server dell’Università di Utah, i quali dispongono delle seguenti caratteristiche:

xl170	200 nodes (Intel Broadwell, 10 core, 1 disk)
CPU	Ten-core Intel E5-2640v4 at 2.4 GHz
RAM	64GB ECC Memory (4x 16 GB DDR4-2400 DIMMs)
Disk	Intel DC S3520 480 GB 6G SATA SSD
NIC	Two Dual-port Mellanox ConnectX-4 25 GB NIC (PCIe v3.0, 8 lanes)

I server sono connessi ad uno switch, pertanto per rimuovere il collegamento diretto tra i due nodi estremi sono state create due VLAN. In questo modo si hanno due reti IP distinte e la possibilità di far partire pacchetti dal *nodo 0* verso il *nodo 2*, e viceversa, facendoli passare per il *nodo 1*, sul quale è caricato il firewall XDP (fig.6).

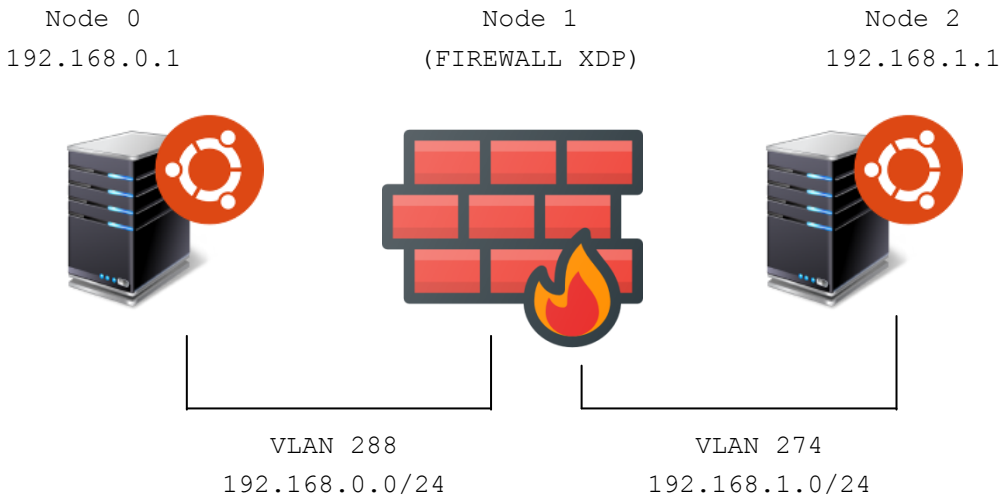

Figura 6. Connessione dei server [6][7].

I dati raccolti sono ottenuti tramite *iperf3,* un tool di Linux che consente di misurare le prestazioni di rete generando un traffico di tipo TCP o UDP tra un *client* e un *server*. Pertanto è stato impostato il *nodo 2* come server in ascolto su una porta:

```
~$ iperf3 -s -p 20239
```

Mentre il *nodo 0* viene impostato come client, generando il traffico, utilizzando per default il protocollo TCP, per 100 secondi:

```
~$ iperf3 -c 192.168.1.1 -p 20239 -t 100
```

Attesi i 100 secondi *iperf3* restituisce la media: dei dati trasmessi, della frequenza di trasmissione e dei pacchetti ritrasmessi.

```
[ ID]Interval           Transfer       Bandwidth         Retr
[  4] 0.00-100.00 sec  180 GBytes     15.4 Gbits/sec   7930    Sender
[  4] 0.00-100.00 sec  180 GBytes     15.4 Gbits/sec           receiver

iperf Done.
```

Il caso illustrato fa riferimento ad una esecuzione dello strumento *iperf3* nella condizione in cui sul *nodo 1*, intermedio, non vi sia caricato alcun programma xdp; in seguito per raccogliere i valori riportati nella *tabella 2* sono stati caricati alternativamente i due tipi diversi di programmi andando ad aumentare il numero di istruzioni utilizzate, da ricordare che in entrambi i modelli di firewall viene controllata la sorgente e la destinazione IP, in particolare i firewall analizzano le porte solo del traffico proveniente dal *nodo 0* diretto al *nodo 2*.

Infine i risultati raccolti in *tabella 2* sono riportanti in figura 7, così da poter mostrare graficamente la differenza tra le prestazioni di banda dei due programmi e il maggior numero di istruzioni nel caso delle mappe, superiore a 10240, altrimenti fermo intorno a 7608.

NUMERO ISTRUZIONI	NO MAPPE			CON MAPPE	
	THR(Gbit/sec)	RETR		THR(Gbit/sec)	RETR
10	11.6	3792		15.1	8187
50	11.6	3443		15.1	9850
500	11.5	4367		15	7535
1024	11.4	4092		15.1	10827
2048	10.9	3659		15.1	12156
3072	10.9	2442		15.2	11766
4096	10.7	2783		15.1	11251
6144	10.2	2699		15.2	12121
7168	10.1	2158		15.1	11358
7608	9.99	2159		15.2	11823
8192	/	/		15.2	10918
10240	/	/		15.1	8303

Tabella 2. Risultati test.

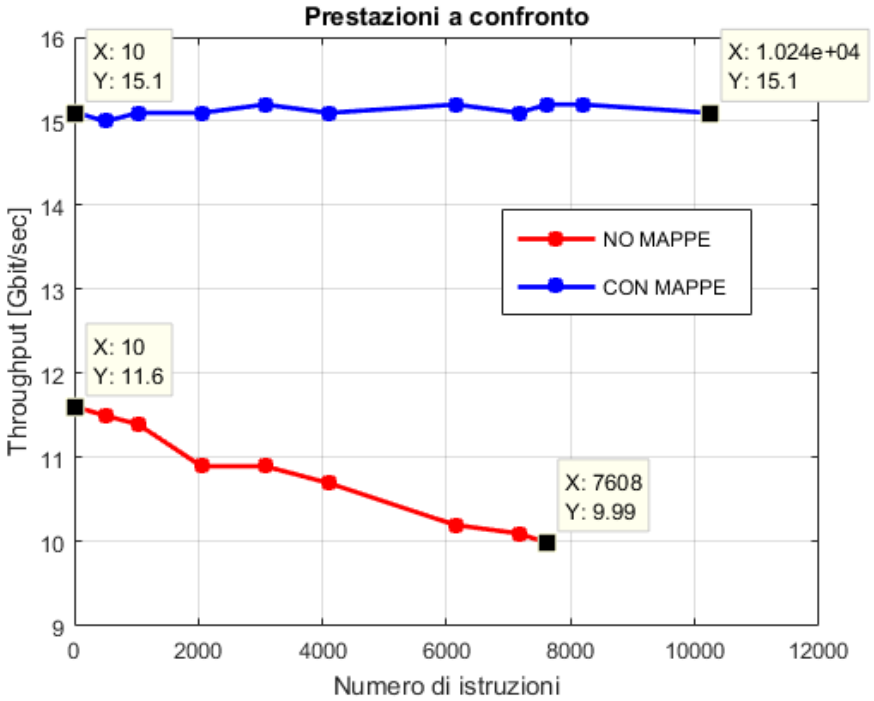

Figura 7. Test su Throughput.

5.3.1 Aumentare le prestazioni dei test

Come emerge dai test effettuati, i programmi le cui istruzioni vengono ricavate attraverso un'indagine di *lookup* da una struttura dati, come la mappa *hash*, risultano più efficienti, in quanto il file contenente il programma XDP presenta un codice maggiormente compatto, privo di numerose istruzioni condizionali.

Pertanto facendo uso delle mappe è possibile ottenere prestazioni ancora migliori se il numero di istruzioni scritte nel codice *C* viene ridotto.

Nel caso considerato è possibile creare per ogni chiave della mappa una struttura con i seguenti campi: indirizzo IP destinazione, IP sorgente e numero di porta. Facendo riferimento all'appendice F, è possibile vedere come ciò è stato implementato, e come viene garantita una riduzione di istruzioni di condizione, dato che il controllo del traffico viene ora effettuato se è trovata la chiave che ha come campi l'indirizzo ip sorgente del *nodo 0* e destinazione quello del *nodo 2*. In questa maniera è stato possibile raggiungere una velocità pari al caso in cui nessun programma sia caricato sull'interfaccia (tabella 3 e figura 8).

NUMERO ISTRUZIONI	**CON MAPPE AVENTI STRUTTURE DI CHIAVI**			**CON MAPPE**	
	THR(Gbit/sec)	RETR		THR(Gbit/sec)	RETR
10	15.4	8501		15.1	8187
50	15.6	8960		15.1	9850
500	15.5	8685		15	7535
1024	15.5	8156		15.1	10827
2048	15.4	8289		15.1	12156
3072	15.6	9989		15.2	11766
4096	15.4	8197		15.1	11251
6144	15.3	9018		15.2	12121
7168	15.6	8104		15.1	11358
7608	15.5	8086		15.2	11823
8192	15.6	8165		15.2	10918
10240	15.5	8482		15.1	8303

Tabella 3. Test sfruttando meglio le mappe.

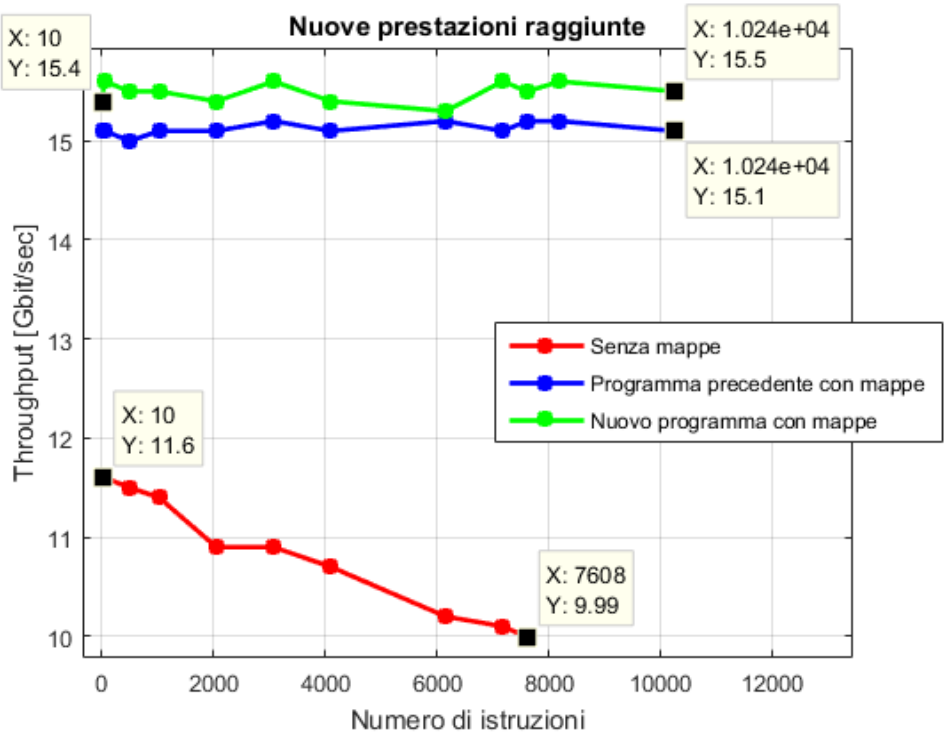

Figura 8. Miglioramento delle prestazioni.

5.4 Limite raggiunto con le mappe

Le mappe hanno permesso al firewall XDP di utilizzare un numero superiore di istruzioni e di compiere un controllo molto più rapido rispetto ai programmi che non ne fanno uso.

Malgrado ciò anch'esse hanno dei limiti, infatti se l'intenzione è quella di creare una mappa avente un numero di istruzioni troppo elevato viene restituito un errore al momento del caricamento del programma, istante in cui viene allocata della memoria per la struttura dati.

```
failed to create a map: 7 Argument list too long
```

Per individuare il limite delle mappe è stato testato il seguente esperimento. Partendo dai codici dei test effettuati nei sottocapitoli precedenti è stato implementato un firewall con il quale viene stressata la mappa hash con un numero di istruzioni dell'ordine del milione. Il firewall va ad analizzare 55535 numeri di porta per ogni indirizzo IP; perciò partendo dall'indirizzo

ip del *nodo 0*[7] sono stati incrementati il numero di indirizzi su cui effettuare il controllo precedentemente spiegato. Il procedimento è stato portato avanti fino al raggiungimento dell'errore in fase di caricamento, ed il numero di istruzioni massimo raggiunto è stato poco più di 42 milioni[8], ovvero per 762 indirizzi ip con l'aggiunta della condizione sulla porta 9999 come per i test precedenti.

Nella tabella 4 e figura 9 vengono riportate le prestazioni all'accrescere degli ingressi della mappa, fino al limite verificato, utilizzando il programma corrispondente al codice nell'appendice G. Come si può notare l'approccio di *lookup* garantisce il medesimo *throughput* costante anche nel caso di una mappa con milioni di ingressi.

L'unico fattore che subisce un cambiamento, e viene incrementato, è il tempo di caricamento del programma sull'interfaccia, dovuto alla creazione della mappa, poiché anche se nello spazio utente, vengono utilizzati due cicli *for* annidati per creare le chiavi ed i corrispondenti valori necessari.

NUMERO DI INDIRIZZI IP	**NUMERO DI ISTRUZIONI**	**THROUGHPUT (Gbit/sec)**	**RETR**
1	55536	15.5	8028
64	3554241	15.4	8750
128	7108481	15.4	7863
254	14105891	15.2	8164
508	28211781	15.5	8657
762	42317671	15.4	9121

Tabella 4. Prestazioni fino al limite consentito.

[7] 192.168.0.1
[8] 1+762*55535 = 42317671

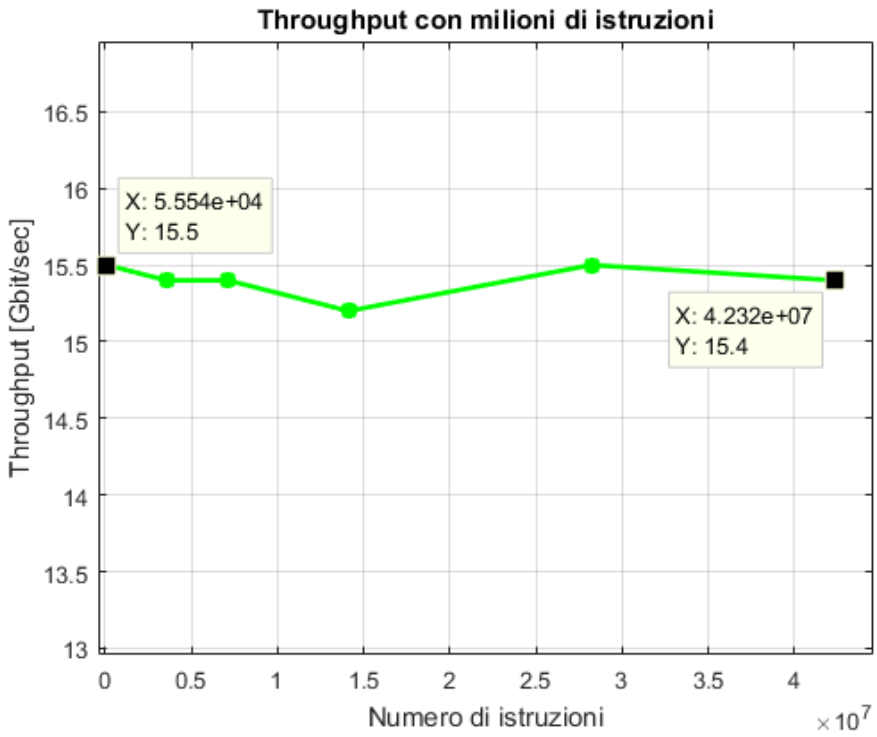

Figura 9. Prestazioni con mappa avente 42 milioni di ingressi.

CONCLUSIONI

Da ciò che è risultato dai test, la possibilità di usare le mappe nei programmi eBPF garantisce enormi vantaggi, tant'è che queste strutture dati possono assicurare, nel caso di un firewall XDP, l'uso di un vasto numero di istruzioni, inserite in maniera rapida tramite l'utilizzo di *cicli for* e allo stesso tempo garantire la possibilità di un processamento veloce all'interno del kernel-space. Questo perché è possibile condividere tutte le informazioni necessarie con il livello utente, garantendo anche l'esecuzione di programmi meno pesanti all'interno del kernel, poiché le istruzioni non vengono scritte nel file "_kern.c" ma richiamate quando necessario dalle mappe tramite funzioni di *sys call*. In particolare come visibile dal grafico in figura 7, il programma che fa uso delle mappe oltre a risultare più veloce mostra un andamento costante all'aumentare delle istruzioni, questo poiché il *lookup* nella mappa hash risulta molto più rapido rispetto al controllo di catene di "if". Infatti nel caso delle mappe, per qualsiasi numero di istruzioni sperimentato, si mantiene circa un *throughput* di *15.1 Gbit/sec*, ovvero quasi come se il programma xdp non fosse presente, condizione in cui la velocità del traffico tra i due nodi risulta circa di *15.4 Gbit/sec*.

Questi ottimi risultati mostrano la possibilità, da parte di XDP, di superare il vantaggio delle tecniche di *kernel bypass* di eseguire l'elaborazione all'interno dell'user-space.

APPENDICE A

Questa appendice fa riferimento al paragrafo 3.2 del capitolo 3, infatti riporta il codice per intero del file "*xdp_drop_user.c"* utilizzato per caricare il programma XDP presente nel file "*xdp_drop_kern.c"*.

```
static const char *__doc__ = "XDP loader\n"
        " - Specify BPF-object --filename to load \n"
        " - and select BPF section --progsec name to XDP-attach to --
dev\n";

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>

#include <bpf/bpf.h>
#include <bpf/libbpf.h>

#include <net/if.h>
#include <linux/if_link.h>

#include "../common/common_params.h"
#include "../common/common_user_bpf_xdp.h"

//Assegna al puntatore "filename" il nome del file di default.
static const char *default_filename = "firewall_10_noMappe_kern.o";

//Assegna il programma di default, nel file, al puntatore "progsec".
static const char *default_progsec = "xdp_firewall_no_mappe";


static const struct option_wrapper long_options[] = {
        {{"help",        no_argument,           NULL, 'h' },
         "Show help", false},

        {{"dev",         required_argument,     NULL, 'd' },
         "Operate on device <ifname>", "<ifname>", true},

        {{"skb-mode",    no_argument,           NULL, 'S' },
         "Install XDP program in SKB (AKA generic) mode"},

        {{"native-mode", no_argument,           NULL, 'N' },
         "Install XDP program in native mode"},

        {{"auto-mode",   no_argument,           NULL, 'A' },
         "Auto-detect SKB or native mode"},

        {{"offload-mode",no_argument,           NULL, 3 },
         "Hardware offload XDP program to NIC"},

        {{"force",       no_argument,           NULL, 'F' },
         "Force install, replacing existing program on interface"},

        {{"unload",      no_argument,           NULL, 'U' },
```

```
         "Unload XDP program instead of loading"},

        {{"quiet",       no_argument,           NULL, 'q' },
         "Quiet mode (no output)"},

        {{"filename",    required_argument,     NULL,  1  },
         "Load program from <file>", "<file>"},

        {{"progsec",    required_argument,      NULL,  2  },
         "Load program in <section> of the ELF file", "<section>"},

        {{0, 0, NULL,  0 }, NULL, false}
};

//Funzione che funziona con oggetto della libreria libbpf;
struct bpf_object *__load_bpf_object_file(const char *filename, int ifin
dex)
{

        int first_prog_fd = -1;
        struct bpf_object *obj;
        int err;

        /* Struttura che permette di impostare il tipo di programma
        *e l'interfaccia su cui caricare il programma.
        */
        struct bpf_prog_load_attr prog_load_attr = {
                .prog_type      = BPF_PROG_TYPE_XDP,
                .ifindex        = ifindex,
        };

//Assegna al campo "file" della seguente struttura come valore il
//nome del file da caricare.
        prog_load_attr.file = filename;

        /* Carica il file oggetto nel kernel
        *via syscall;
        */
       err = bpf_prog_load_xattr(&prog_load_attr, &obj, &first_prog_fd);
       if (err) {
             fprintf(stderr, "ERR: loading BPF-OBJ file(%s) (%d): %s\n",
                        filename, err, strerror(-err));
                return NULL;
        }

        /* restituisce il puntatore */
        return obj;
}

struct bpf_object *__load_bpf_and_xdp_attach(struct config *cfg)
{

        struct bpf_program *bpf_prog;
        struct bpf_object *bpf_obj;
        int offload_ifindex = 0;
        int prog_fd = -1;
        int err;


        if (cfg->xdp_flags & XDP_FLAGS_HW_MODE)
                offload_ifindex = cfg->ifindex;
```

```
//Carica file oggetto nel kernel e lo restituisce a bpf_obj;
   bpf_obj = __load_bpf_object_file(cfg->filename, offload_ifindex);
       if (!bpf_obj) {
              fprintf(stderr, "ERR: loading file: %s\n", cfg->filename);
                exit(EXIT_FAIL_BPF);
        }


// Usata funzione per trovare programma (SEC) in file (oggetto bpf_obj).
   bpf_prog = bpf_object__find_program_by_title(bpf_obj, cfg->progsec);
        if (!bpf_prog) {
            fprintf(stderr, "ERR: finding progsec: %s\n", cfg->progsec);
                exit(EXIT_FAIL_BPF);
        }

//Funzione per ottenere descrittore (rappresentatore del file) del file
//da collegare all'hook XDP.
        prog_fd = bpf_program__fd(bpf_prog);
        if (prog_fd <= 0) {           //Se minore di zero c'è un errore.
                fprintf(stderr, "ERR: bpf_program__fd failed\n");
                exit(EXIT_FAIL_BPF);
        }

        /*Per caricare il programma viene passato il suo file
        *descrittore e l'interfaccia su cui deve essere caricato
        */
        err = xdp_link_attach(cfg->ifindex, cfg->xdp_flags, prog_fd);
        if (err)
                exit(err);

        return bpf_obj;
}
//Stampa la lista dei programmi disponibili nel file;
static void list_avail_progs(struct bpf_object *obj)
{
        struct bpf_program *pos;

        printf("BPF object (%s) listing avail --progsec names\n",
               bpf_object__name(obj));

        bpf_object__for_each_program(pos, obj) {
                if (bpf_program__is_xdp(pos))
                        printf(" %s\n", bpf_program__title(pos, false));
        }
}

// int argc: contiene il numero di stringhe inserite dall'utente a linea
// di comando;
// char *argv[]: l'array che contiene le stringhe
// inserite dall'utente a linea di comando,
//(ogni elemento dell'array è un puntatore a carattere).
int main(int argc, char **argv)
{
     struct bpf_object *bpf_obj;

     struct config cfg = {
          .xdp_flags = XDP_FLAGS_UPDATE_IF_NOEXIST | XDP_FLAGS_DRV_MODE,
               .ifindex   = -1,
               .do_unload = false,
        };
```

```
//Copia il nome del file oggetto nel campo della struttura config;
  strncpy(cfg.filename, default_filename, sizeof(cfg.filename));

//Copia il nome del programma;
  strncpy(cfg.progsec,  default_progsec,  sizeof(cfg.progsec));

/* Funzione per monitorare i comandi inseriti dalla command line */
   parse_cmdline_args(argc, argv, long_options, &cfg, __doc__);

        /* Opzioni di controllo */
        if (cfg.ifindex == -1) {
                fprintf(stderr, "ERR: required option --dev missing\n");
                usage(argv[0], __doc__, long_options, (argc == 1));
                return EXIT_FAIL_OPTION;
        }
      /*Se "do_unload" è diverso da zero e quindi vero viene restituita
       *la funzione che stacca il programma dall'interfaccia;
       */
        if (cfg.do_unload)
                return xdp_link_detach(cfg.ifindex, cfg.xdp_flags, 0);

        //Funzione sopra definita per caricare il programma;
        bpf_obj = __load_bpf_and_xdp_attach(&cfg);
        if (!bpf_obj)
                return EXIT_FAIL_BPF;

if (verbose)
  //Fuzione che fa stampare a video le opzioni del programma caricato;
                list_avail_progs(bpf_obj);

if (verbose) {
    //Ciò che viene stampato a video in caso di successo del caricamento
 //del programma.
        printf("Success: Loaded BPF-object(%s) and used section(%s)\n",
                       cfg.filename, cfg.progsec);
        printf(" - XDP prog attached on device:%s(ifindex:%d)\n",
                       cfg.ifname, cfg.ifindex);
        }

   /* Se tutto è andato a buon fine viene restituito il main restituisce
 zero; */
        return EXIT_OK;
}
```

APPENDICE B

L'appendice illustra un esempio di programma eBPF di tipo Tracepoint; tal esempio è presente tra quelli di xdp-tutorial, infatti, utilizzando il metodo proposto da quest'ultimo, vi sono due file uno dei quali utile per il caricamento, anche se non essendo un programma xdp non viene caricato su un'interfaccia, ma si occupa di monitorarne il traffico, pertanto può essere caricato anche se già presente un XDP. In particolare il programma di tracciamento sfrutta l'utilizzo della mappa "BPF_MAP_TYPE_PERCPU_HASH" per tenere il conto dei pacchetti scartati, dai vari processori, passanti per l'interfaccia desiderata.

trace_prog_kern.c :

```
#include <linux/bpf.h>
#include <bpf/bpf_helpers.h>

struct bpf_map_def SEC("maps") xdp_stats_map = {
    .type        = BPF_MAP_TYPE_PERCPU_HASH,
    .key_size    = sizeof(__s32),
    .value_size  = sizeof(__u64),
    .max_entries = 10,
};

struct xdp_exception_ctx {
    __u64 __pad;      // Primi 8 bytes non accessibili da bpf code
    __s32 prog_id;    //      offset:8;  size:4; signed:1;
    __u32 act;        //      offset:12; size:4; signed:0;
    __s32 ifindex;    //      offset:16; size:4; signed:1;
};
 //Sezione in cui deve essere memorizzato il  programma di Tracepoint;
SEC("tracepoint/xdp/xdp_exception")
int trace_xdp_exception(struct xdp_exception_ctx *ctx)
{
    //Assegna come chiave l'indice dell'interfaccia usata;
    __s32 key = ctx->ifindex;
    __u32 *valp;

    /*Se l'azione è "XDP_ABORTED" allora il programma
    * contiuna per incrementate i pacchetti scartati;
    */
    if (ctx->act != XDP_ABORTED)
        return 0;
```

```
    //Prende il valore di pacchetti scartati da quell'interfaccia;
    valp = bpf_map_lookup_elem(&xdp_stats_map, &key);

    /*Se non è stato scartato ancora nessun pacchetto
    * inizializza a "1" il valore degli scarti;
    */
    if (!valp) {
        __u64 one = 1;
        return bpf_map_update_elem(&xdp_stats_map, &key, &one, 0) ? 1 : 0
;
    }
    //Altrimenti incrementa il valore;
    (*valp)++;
    return 0;
}

char _license[] SEC("license") = "GPL";
```

trace_load_and_stats.c :

```
static const char *__doc__ = "XDP loader and stats program\n"
    " - Allows selecting BPF section --progsec name to XDP-attach to --
dev\n";

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <getopt.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <net/if.h>

#include <locale.h>
#include <unistd.h>
#include <time.h>

#include <bpf/bpf.h>
#include <bpf/libbpf.h>

#include <net/if.h>
#include <linux/if_link.h>

#include <linux/err.h>

#include "../common/common_params.h"
#include "../common/common_user_bpf_xdp.h"
#include "../common/common_libbpf.h"
#include "bpf_util.h"
```

```
#include <linux/perf_event.h>
#define _GNU_SOURCE
#include <unistd.h>
#include <sys/syscall.h>
#include <sys/ioctl.h>

#ifndef PATH_MAX
#define PATH_MAX    4096
#endif

static const char *default_filename = "trace_prog_kern.o";

static const struct option_wrapper long_options[] = {
    {{"help",        no_argument,       NULL, 'h' },
     "Show help", false},

    {{"quiet",       no_argument,       NULL, 'q' },
     "Quiet mode (no output)"},

    {{"filename",    required_argument, NULL,  1  },
     "Load program from <file>", "<file>"},

    {{0, 0, NULL,  0 }}
};

// Funzione per ricavare il descrittore della mappa;
int find_map_fd(struct bpf_object *bpf_obj, const char *mapname)
{
    struct bpf_map *map;
    // Inizialmente impostata nessuna mappa presente;
    int map_fd = -1;

    /*Utilizzando la seguente "helper function"
    * è possibile ricavarsiil nome della mappa
    * presente nel file oggetto
    */
    map = bpf_object__find_map_by_name(bpf_obj, mapname);
        if (!map) {
        fprintf(stderr, "ERR: cannot find map by name: %s\n", mapname);
        //Se non c'è alcuna mappa il programma esce dalla funzione;
        goto out;
    }

    /* Funzione che restituisce il descrittore
    * della mappa indicata;
    */
    map_fd = bpf_map__fd(map);
 out:
    return map_fd;
```

```
}

static void stats_print(int map_fd)
{
    //Prende il numero di cpu;
    unsigned int nr_cpus = bpf_num_possible_cpus();
    __u64 values[nr_cpus];
    __s32 key;
    void *keyp = &key, *prev_keyp = NULL;
    int err;

    // Ciclo infinito
    while (true) {
        char dev[IF_NAMESIZE];
        __u64 total = 0;
        int i;

        // Finchè ci sono chiavi il ciclo continua;
        err = bpf_map_get_next_key(map_fd, prev_keyp, keyp);
        if (err) {
            /*Se la chiave l'ultima err == -1,
            * e viene restituito il seguente errore;
            */
            if (errno == ENOENT)
                err = 0;
            //Esce dal ciclo "while";
            break;
        }

        if ((bpf_map_lookup_elem(map_fd, keyp, values)) != 0) {
            fprintf(stderr,
                "ERR: bpf_map_lookup_elem failed key:0x%X\n", key);
        }

        /* Somma i valori da ogni CPU */
        for (i = 0; i < nr_cpus; i++)
        //Somma il numero di pacchetti scartati da tutte le CPU;
        total += values[i];

//"if_indextoname"--> dall'indice dell'interfaccia si ricava in nome;
        printf("%s (%llu) ", if_indextoname(key, dev), total);

        //La chiave corrente diventa la precedente;
        prev_keyp = keyp;
        }

    printf("\n");
}

static void stats_poll(int map_fd, __u32 map_type, int interval)
```

```
{
    setlocale(LC_NUMERIC, "en_US");

    while (1) {
        stats_print(map_fd);
        sleep(interval);
    }
}

// Controllo funzionamento mappa;
static int __check_map_fd_info(int map_fd, struct bpf_map_info *info,
                   struct bpf_map_info *exp)
{
    __u32 info_len = sizeof(*info);
    int err;

    if (map_fd < 0)
        return EXIT_FAIL;

        /* BPF-info via bpf-syscall */
    err = bpf_obj_get_info_by_fd(map_fd, info, &info_len);
    if (err) {
        fprintf(stderr, "ERR: %s() can't get info - %s\n",
            __func__,  strerror(errno));
        return EXIT_FAIL_BPF;
    }

    if (exp->key_size && exp->key_size != info->key_size) {
        fprintf(stderr, "ERR: %s() "
            "Map key size(%d) mismatch expected size(%d)\n",
            __func__, info->key_size, exp->key_size);
        return EXIT_FAIL;
    }
    if (exp->value_size && exp->value_size != info->value_size) {
        fprintf(stderr, "ERR: %s() "
            "Map value size(%d) mismatch expected size(%d)\n",
            __func__, info->value_size, exp->value_size);
        return EXIT_FAIL;
    }
    if (exp->max_entries && exp->max_entries != info->max_entries) {
        fprintf(stderr, "ERR: %s() "
            "Map max_entries(%d) mismatch expected size(%d)\n",
            __func__, info->max_entries, exp->max_entries);
        return EXIT_FAIL;
    }
    if (exp->type && exp->type  != info->type) {
        fprintf(stderr, "ERR: %s() "
            "Map type(%d) mismatch expected type(%d)\n",
            __func__, info->type, exp->type);
        return EXIT_FAIL;
```

```
    }

    return 0;
}

int filename__read_int(const char *filename, int *value)
{
    char line[64];
    //"open(..., O_RDONLY)" ---> apre il file per la lettura;
    int fd = open(filename, O_RDONLY), err = -1;

    if (fd < 0)
        return -1;
  //"read"---> Legge da "fd" la dimensione di line, memorizza in line;
    if (read(fd, line, sizeof(line)) > 0) {

        //atoi ---> converte una stringa in intero;
        *value = atoi(line);
        err = 0;
    }

    close(fd);
    return err;
}
//Percorso a cui collegare il programma di tracepoint;
#define TP "/sys/kernel/debug/tracing/events/"

static int read_tp_id(const char *name, int *id)
{
    char path[PATH_MAX];
    /*Al TP viene aggiunto il sottosistema
    * e il nome del tracepoint (nome della sezione); */
    snprintf(path, PATH_MAX, TP "%s/id", name);
    return filename__read_int(path, id);
}

static inline int
sys_perf_event_open(struct perf_event_attr *attr,
            pid_t pid, int cpu, int group_fd,
            unsigned long flags)
{
 return syscall(__NR_perf_event_open, attr, pid, cpu, group_fd, flags);
}

static struct bpf_object* load_bpf_and_trace_attach(struct config *cfg)
{
    struct perf_event_attr attr;
    struct bpf_object *obj;
    int err, bpf_fd;
    int id, fd;
```

```
//Carica il programma eBPF specificandone il tipo;
if(bpf_prog_load(cfg->filename,
 BPF_PROG_TYPE_TRACEPOINT, &obj, &bpf_fd)) {
        fprintf(stderr, "ERR: failed to load program\n");
        goto err;
    }

    if (read_tp_id("xdp/xdp_exception", &id)) {
        fprintf(stderr, "ERR: can't get program section\n");
        goto err;
    }

    /* Imposta tracepoint perf event,
    * inizializzando inizilamente tutto
    * a zero;
    */
    memset(&attr, 0, sizeof(attr));
    attr.type = PERF_TYPE_TRACEPOINT;
    attr.config = id;
    attr.sample_period = 1;

    /* apre perf even */
    fd = sys_perf_event_open(&attr, -1, 0, -1, 0);
    if (fd <= 0) {
        fprintf(stderr, "ERR: failed to open perf event %s\n",
            strerror(errno));
        goto err;
    }

    // Assegna il programma al tracepoint perf event,
    err = ioctl(fd, PERF_EVENT_IOC_SET_BPF, bpf_fd);
    if (err) { fprintf(stderr, "ERR: failed to
                      connect perf event with eBPF prog %s\n",
                      strerror(errno));
        goto err;
    }

    // Abilita l'evento;
    err = ioctl(fd, PERF_EVENT_IOC_ENABLE, 0);
    if (err) {
        fprintf(stderr, "ERR: failed to enable perf event %s\n",
            strerror(errno));
        goto err;
    }

    return obj;

err:
    bpf_object__close(obj);
    return NULL;
```

```
}

int main(int argc, char **argv)
{
    struct bpf_map_info map_expect = { 0 };
    struct bpf_map_info info = { 0 };
    struct bpf_object *bpf_obj;
    struct config cfg;
    int stats_map_fd;
    int interval = 2;
    int err;

    //Programma che carica è quello di tracepoint;
    strncpy(cfg.filename, default_filename, sizeof(cfg.filename));

    parse_cmdline_args(argc, argv, long_options, &cfg, __doc__);

    bpf_obj = load_bpf_and_trace_attach(&cfg);
    //Se non c'è nessun file oggetto viene restituito un errore;
    if (!bpf_obj)
        return EXIT_FAIL_BPF;

    if (verbose) {
        printf("Success: Loaded BPF-object(%s)\n", cfg.filename);
    }
    //Assegno identificatore della mappa cercata,
    // se trovata, nel file oggetto;
    stats_map_fd = find_map_fd(bpf_obj, "xdp_stats_map");

    if (stats_map_fd < 0)
        return EXIT_FAIL_BPF;

    map_expect.key_size    = sizeof(__s32);
    map_expect.value_size  = sizeof(__u64);

    //Controlla che dimensioni siano quelle aspettate;
    err = __check_map_fd_info(stats_map_fd, &info, &map_expect);
    if (err) {
        fprintf(stderr, "ERR: map via FD not compatible\n");
        return err;
    }
    //Se mappa è caricata correnttamente
    //ne stampa le caratteristiche a video;
    if (verbose) {
        printf("\nCollecting stats from BPF map\n");
        printf(" - BPF map (bpf_map_type:%d) id:%d name:%s"
               " key_size:%d value_size:%d max_entries:%d\n",
               info.type, info.id, info.name,
               info.key_size, info.value_size, info.max_entries
               );
```

```
    }
    //Stampa stato della mappa;
    stats_poll(stats_map_fd, info.type, interval);
    return EXIT_OK;
}
```

APPENDICE C

Viene proposto un esempio di programma "*socket_filter*", il quale, tra le mappe utilizzate, fa uso della "BPF_MAP_TYPE_PROG_ARRAY" e della funzione "*bpf_tail_call"* per saltare al programma più opportuno in base al tipo di pacchetto da analizzare. I codici riportati con i dovuti commenti fanno riferimento all'esempio "*sockex3"* presente in "*sample/bpf"*, pertanto come spigato nella tesi il programma viene caricato in maniera diversa rispetto agli altri casi visti in precedenza. Infatti, per caricare un programma basta passare il nome del file "_kern.o" alla funzione "*load_bpf_file"*, la quale è richiamata nel file "_user.c" e definita in "load_bpf.c". Inoltre altra comodità di quest'approccio è che non occorre specificare il tipo di programma eBPF poiché in base al nome della sezione, "load_bpf.o" si comporta di conseguenza. Infine vi è la presenza di array, definiti nelle librerie, nei quali vengono memorizzati i descrittori delle mappe e dei programmi presenti in "_kern.c", in questo modo è più semplice richiamare una mappa da utilizzare.

sockex3_test_kern.c :

```
#include <uapi/linux/bpf.h>
#include "bpf_helpers.h"
#include <uapi/linux/in.h>
#include <uapi/linux/if.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/ipv6.h>
#include <uapi/linux/if_tunnel.h>
#include <uapi/linux/mpls.h>
#define IP_MF       0x2000
#define IP_OFFSET   0x1FFF

#define PROG(F) SEC("socket/"__stringify(F)) int bpf_func_##F

struct bpf_map_def SEC("maps") jmp_table = {
    .type = BPF_MAP_TYPE_PROG_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(u32),
    //Permette di caricare 8 programmi diversi;
    .max_entries = 8,
};
```

```
#define PARSE_VLAN 1
#define PARSE_MPLS 2
#define PARSE_IP 3
#define PARSE_IPV6 4

/*In base all'intestazione ethernet
* salta al programma desiderato,
* grazie a bpf_tail_call;
*/
static inline void parse_eth_proto(struct __sk_buff *skb, u32 proto)
{
    switch (proto) {
    case ETH_P_8021Q:
    case ETH_P_8021AD:
        bpf_tail_call(skb, &jmp_table, PARSE_VLAN);
        break;
    case ETH_P_MPLS_UC:
    case ETH_P_MPLS_MC:
        bpf_tail_call(skb, &jmp_table, PARSE_MPLS);
        break;
    case ETH_P_IP:
        bpf_tail_call(skb, &jmp_table, PARSE_IP);
        break;
    case ETH_P_IPV6:
        bpf_tail_call(skb, &jmp_table, PARSE_IPV6);
        break;
    }
}

struct vlan_hdr {
    __be16 h_vlan_TCI;
    __be16 h_vlan_encapsulated_proto;
};

struct bpf_flow_keys {
    __be32 src;
    __be32 dst;
    union {
        __be32 ports;
        __be16 port16[2];
    };
    __u32 ip_proto;
};

static inline int ip_is_fragment(struct __sk_buff *ctx, __u64 nhoff)
{
    return load_half(ctx, nhoff + offsetof(struct iphdr, frag_off))
        & (IP_MF | IP_OFFSET);
}
```

```
static inline __u32 ipv6_addr_hash(struct __sk_buff *ctx, __u64 off)
{
    __u64 w0 = load_word(ctx, off);
    __u64 w1 = load_word(ctx, off + 4);
    __u64 w2 = load_word(ctx, off + 8);
    __u64 w3 = load_word(ctx, off + 12);

    return (__u32)(w0 ^ w1 ^ w2 ^ w3);
}

struct globals {
    /*Variabile di tipo "struct bpf_flow_keys";
    * usata per assegnare valori alle chiavi; */
    struct bpf_flow_keys flow;
};

struct bpf_map_def SEC("maps") percpu_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(struct globals),
    .max_entries = 32,
};

static struct globals *this_cpu_globals(void)
{
    u32 key = bpf_get_smp_processor_id();

    return bpf_map_lookup_elem(&percpu_map, &key);
}

/*Struttura che contiene variabili da stampare,
* condivisa tra user e kern; */
struct pair {
    __u64 packets;
    __u64 bytes;
};

struct bpf_map_def SEC("maps") hash_map = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = sizeof(struct bpf_flow_keys),
    .value_size = sizeof(struct pair),
    .max_entries = 1024,
};

static void update_stats(struct __sk_buff *skb, struct globals *g)
{
    //Punta alle chiavi;
    struct bpf_flow_keys key = g->flow;
    struct pair *value;
```

```
    //Prendo valore corrispondente a certa chiave;
    value = bpf_map_lookup_elem(&hash_map, &key);

    //Se valore esiste;
    if (value) {
        //Incrementa numero pacchetti di 1;

        __sync_fetch_and_add(&value->packets, 1);

 //Incrementa numero di bytes che prende
 //da lunghezza socket (pacchetto);
        __sync_fetch_and_add(&value->bytes, skb->len);
    } else {
        //Altrimenti lo crea, un solo pacchetto;
        struct pair val = {1, skb->len};

        //Aggiorna mappa con valori ottenuti;
        bpf_map_update_elem(&hash_map, &key, &val, BPF_ANY);
    }
}

static __always_inline void parse_ip_proto(struct __sk_buff *skb,
                      struct globals *g, __u32 ip_proto)
{
    __u32 nhoff = skb->cb[0];
    int poff;

    switch (ip_proto) {
    case IPPROTO_GRE: {
        struct gre_hdr {
            __be16 flags;
            __be16 proto;
        };

        __u32 gre_flags = load_half(skb,
                        nhoff + offsetof(struct gre_hdr, flags));
        __u32 gre_proto = load_half(skb,
                        nhoff + offsetof(struct gre_hdr, proto));

        if (gre_flags & (GRE_VERSION|GRE_ROUTING))
            break;

        nhoff += 4;
        if (gre_flags & GRE_CSUM)
            nhoff += 4;
        if (gre_flags & GRE_KEY)
            nhoff += 4;
        if (gre_flags & GRE_SEQ)
            nhoff += 4;
```

```
            skb->cb[0] = nhoff;
            parse_eth_proto(skb, gre_proto);
            break;
        }
        case IPPROTO_IPIP:
            parse_eth_proto(skb, ETH_P_IP);
            break;
        case IPPROTO_IPV6:
            parse_eth_proto(skb, ETH_P_IPV6);
            break;
        case IPPROTO_TCP:
        case IPPROTO_UDP:
            g->flow.ports = load_word(skb, nhoff);
        case IPPROTO_ICMP:
            g->flow.ip_proto = ip_proto;
            update_stats(skb, g);     //Definita sopra;
            break;
        default:
            break;
        }
}

PROG(PARSE_IP)(struct __sk_buff *skb)
{
    struct globals *g = this_cpu_globals();
    __u32 nhoff, verlen, ip_proto;

    if (!g)
        return 0;

    nhoff = skb->cb[0];

    if (unlikely(ip_is_fragment(skb, nhoff)))
        return 0;

    //Carica campo IP protocol;
    ip_proto = load_byte(skb, nhoff + offsetof(struct iphdr, protocol));

  //Se campo protocol è diverso da GRE (incapsulamento di IP dentro IP);

    if (ip_proto != IPPROTO_GRE) {

 //Nella chiave src mette indirizzo sorgente;
 // (offsetof---> salta al campo della struttura);
  g->flow.src = load_word(skb, nhoff + offsetof(struct iphdr, saddr));

 //Nella chiave dst mette indirizzo destinazione;
  g->flow.dst = load_word(skb, nhoff + offsetof(struct iphdr, daddr));
    }
```

```
//Memorizzo versione e lunghezza;
    verlen = load_byte(skb, nhoff + 0/*offsetof(struct iphdr, ihl)*/);

    /*Tengo solo la lunghezza e con shiftt di 2
    *a sinistra moltiplico per 4,
    *(32 bit) ottenendo così la lunghezza cdell'intero pacchetto;
    *aggiungo sempre lunghezza per poter puntare
    *al pacchetto succhessivo ogni volta;
    */
    nhoff += (verlen & 0xF) << 2;
    skb->cb[0] = nhoff;

   /*Passo puntatore al pacchetto, puntatore a struttura globale,
   * e campo ip_proto alla funzione che andrà a elaborare componente ip;
    */
    parse_ip_proto(skb, g, ip_proto);
    return 0;
}

PROG(PARSE_IPV6)(struct __sk_buff *skb)
{
    struct globals *g = this_cpu_globals();
    __u32 nhoff, ip_proto;

    if (!g)
        return 0;

    nhoff = skb->cb[0];

    ip_proto = load_byte(skb,
                nhoff + offsetof(struct ipv6hdr, nexthdr));
    g->flow.src = ipv6_addr_hash(skb,
                    nhoff + offsetof(struct ipv6hdr, saddr));
    g->flow.dst = ipv6_addr_hash(skb,
                    nhoff + offsetof(struct ipv6hdr, daddr));
    nhoff += sizeof(struct ipv6hdr);

    skb->cb[0] = nhoff;
    parse_ip_proto(skb, g, ip_proto);
    return 0;
}

PROG(PARSE_VLAN)(struct __sk_buff *skb)
{
    __u32 nhoff, proto;

    nhoff = skb->cb[0];

    proto = load_half(skb, nhoff + offsetof(struct vlan_hdr,
                        h_vlan_encapsulated_proto));
```

```
    nhoff += sizeof(struct vlan_hdr);
    skb->cb[0] = nhoff;

    parse_eth_proto(skb, proto);

    return 0;
}

PROG(PARSE_MPLS)(struct __sk_buff *skb)
{
    __u32 nhoff, label;

    nhoff = skb->cb[0];

    label = load_word(skb, nhoff);
    nhoff += sizeof(struct mpls_label);
    skb->cb[0] = nhoff;

    if (label & MPLS_LS_S_MASK) {
        __u8 verlen = load_byte(skb, nhoff);
        if ((verlen & 0xF0) == 4)
            parse_eth_proto(skb, ETH_P_IP);
        else
            parse_eth_proto(skb, ETH_P_IPV6);
    } else {
        parse_eth_proto(skb, ETH_P_MPLS_UC);
    }

    return 0;
}

SEC("socket/0")
//"skb" puntatore al buffer dove c'è il pacchetto (socket buffer);
int main_prog(struct __sk_buff *skb)
{
    //Salva tutto Header del frame ethernet (14 bytes);
    __u32 nhoff = ETH_HLEN;

    /*12 bytes (destination+source) da saltare,
    * così restano i 2 bytes del MAC header (Ethernet type),
    *memorizzati in proto;
    */
    __u32 proto = load_half(skb, 12);

 //Salva Header ethernet dentro
 //control buffer (nhoff = network ethernet offset),
 //e man mano aggiunge lunghezza dei pacchetti;
    skb->cb[0] = nhoff;

    //Passa puntatore al campo type e il campo type,
```

```
    //per capire cosa sta trasportando la trama ethernet;
    parse_eth_proto(skb, proto);
    return 0;
}

char _license[] SEC("license") = "GPL";
```

sockex3_test_user.c

```
// SPDX-License-Identifier: GPL-2.0
#include <stdio.h>
#include <assert.h>
#include <linux/bpf.h>
#include <bpf/bpf.h>
#include "bpf_load.h"
#include "sock_example.h"
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/resource.h>

#define PARSE_IP 3
#define PARSE_IP_PROG_FD (prog_fd[0])
#define PROG_ARRAY_FD (map_fd[0])

//Struttura contenente le chiavi;
struct bpf_flow_keys {
    __be32 src;
    __be32 dst;
    union {
        //Elemento unico;
        __be32 ports;
        //Array di due elementi da 16 bit;
        __be16 port16[2];
    };
    __u32 ip_proto;
};

//Struttura che contiene variabili da stampare;
struct pair {
    __u64 packets;
    __u64 bytes;
};

int main(int argc, char **argv)
{
    struct rlimit r = {RLIM_INFINITY, RLIM_INFINITY};
    char filename[256];
    FILE *f;
```

```
/*Terza chiave fa riferimento al programma che analizza
* pacchetti con IP;
*/
int i, sock, err, id, key = PARSE_IP;
struct bpf_prog_info info = {};
uint32_t info_len = sizeof(info);

//Copia il nome del file nell'array "filename";
snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);
setrlimit(RLIMIT_MEMLOCK, &r);

//Carica il programma;
if (load_bpf_file(filename)) {
    //Se c'è un errore lo stampa;
    printf("%s", bpf_log_buf);
    return 1;
}

//Test per verificare che sia caricato correttamante;
err = bpf_obj_get_info_by_fd(PARSE_IP_PROG_FD, &info, &info_len);
assert(!err);
err = bpf_map_lookup_elem(PROG_ARRAY_FD, &key, &id);
assert(!err);
assert(id == info.id);

//Apertura della socket sull'interfaccia ens1f1;
sock = open_raw_sock("ens1f1");

/*imposta opzione socket per far funzionare
* il quarto programma eBPF definito nel file "_kern.c";
*/
assert(setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd[4],
          sizeof(__u32)) == 0);

//Se dato un argomento qualsiasi, da terminale, fa il "ping";
if (argc > 1)
    f = popen("ping -c5 192.168.1.1", "r");
else
    f = popen("netperf -l 4 localhost", "r");
(void) f;

//Stampa informazioni per 5 volte;
for (i = 0; i < 5; i++) {
    struct bpf_flow_keys key = {}, next_key;

    //Variabile con cui accede alla struttura "pair";
    struct pair value;

    /*Sospende il processo corrente per un secondo
    * o fino all'arrivo di un segnale che non sia ignorato;
```

```
        */
        sleep(1);    //La funzione sleep(), in #include <unistd.h>;

        //Stampa valori che prenderà dalla mappa "hash_map"... map_fd=2;

printf("IP    src.port -> dst.port              bytes      packets\n");

        //Legge la prossima chiave della mappa corrispondente;
        while (bpf_map_get_next_key(map_fd[2], &key, &next_key) == 0) {


            //Restituisce in "value" il valore della mappa
            //corrispondente alla chiave passata;
            bpf_map_lookup_elem(map_fd[2], &next_key, &value);

            printf("%s.%05d -> %s.%05d %12lld %12lld\n",

            //Converte indirizzi (IPv4) in stringhe di numeri decimali;
                   inet_ntoa((struct in_addr){htonl(next_key.src)}),
                   next_key.port16[0],
                   inet_ntoa((struct in_addr){htonl(next_key.dst)}),
                   next_key.port16[1],
                   value.bytes, value.packets);
            key = next_key;
        }
close(sock);
    }
    return 0;
}
```

APPENDICE D

Viene riportato il codice del file *firewall_10_noMappe_kern.c*, utilizzato per i test senza mappe, il quale contiene 10 istruzioni. Il file "_user.c" non è riportato poiché identico a quello presente nell'appendice A, dal quale occorre cambiare il nome del file oggetto e della sezione da caricare con le seguenti: "*firewall_10_noMappe_kern.o" e "xdp_firewall_no_mappe".*

```
#include <stddef.h>
#include <linux/bpf.h>
#include <linux/in.h>
#include <linux/if_ether.h>
#include <linux/if_packet.h>
#include <linux/ipv6.h>
#include <linux/ip.h>
#include <linux/icmpv6.h>
#include <linux/icmp.h>
#include <bpf/bpf_helpers.h>
#include <bpf/bpf_endian.h>
//Contiene funzioni per elaborare vari campi del pacchetto;
#include "../common/parsing_helpers.h"
#include <linux/udp.h>
#include <linux/tcp.h>

//IP nodo 0;
#define IP0 3232235521
//IP nodo 2;
#define IP2 3232235777

SEC("xdp_firewall_no_mappe")
int  xdp_firewall(struct xdp_md *ctx)
{

    __u16 port_value=0;
    __u32 ip_src;
    __u32 ip_dest;

    int eth_type, ip_type;
        struct ethhdr *eth;
        struct iphdr *iph;

        struct udphdr *udp;
        struct tcphdr *tcp;
        void *data_end = (void *)(long)ctx->data_end;
        void *data = (void *)(long)ctx->data;
        struct hdr_cursor nh = { .pos = data };
```

```
        eth_type = parse_ethhdr(&nh, data_end, &eth);
        if (eth_type < 0) {
                return XDP_ABORTED;
        }

        if (eth_type == bpf_htons(ETH_P_IP)) {
                ip_type = parse_iphdr(&nh, data_end, &iph);

                 if(ip_type < 0) {
                        return XDP_ABORTED;
                }

      ip_src = bpf_ntohl(iph->saddr);
      ip_dest = bpf_ntohl(iph->daddr);

    if ( ip_src == (__u32)IP0 && ip_dest == (__u32)IP2 ) {
        if (ip_type == IPPROTO_TCP) {

        //Se lunghezza tcp minore di zero;
        if ( parse_tcphdr(&nh, data_end, &tcp) < 0 ) {
            return XDP_ABORTED;
        }

        port_value = bpf_ntohs(tcp->dest);
        goto out;
    }

        if ( ip_type == IPPROTO_UDP ) {

         //Se lunghezza udp minore di zero;
        if ( parse_udphdr(&nh, data_end, &udp) < 0 ) {
            return XDP_ABORTED;
        }
        port_value = bpf_ntohs(udp->dest);
        goto out;
        }
    }
    return XDP_PASS;
}
return XDP_PASS;

//Ci arriva solo se è un protocollo TCP o UDP;
out:
//N.B obiettivo è confrontare più porte possibili
//allungando numero istruzioni;

if( port_value == 18657 )
        goto out2;
if( port_value == 28306 )
```

```
        goto out2;
if( port_value == 27542 )
        goto out2;
if( port_value == 27722 )
        goto out2;
if( port_value == 11018 )
        goto out2;
if( port_value == 26030 )
        goto out2;
if( port_value == 15023 )
        goto out2;
if( port_value == 22500 )
        goto out2;
if( port_value == 29317 )
        goto out2;

if( port_value == 9999 ) {

        return XDP_DROP;
}

out2:

//Controllo porte dice che può passare;
return XDP_PASS;


}

char _license[] SEC("license") = "GPL";
```

APPENDICE E

Nel caso con le mappe è riportato l'esempio con 10240 istruzioni, dato che creando la mappa con un ciclo *for* non si ottiene un codice troppo lungo.

firewall_10240_user.c :

```
#include <linux/bpf.h>
#include <linux/if_link.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <arpa/inet.h>
#include <netinet/ether.h>
#include <unistd.h>
#include <time.h>
#include "bpf_load.h"
#include <bpf/bpf.h>
#include "bpf_util.h"

// Assegna l'indice dell'interfaccia ens1f1;
static int ifindex = 5;
static __u32 xdp_flags = 0;

//Funzione chiamata per interrompere il file kern;
static void int_exit(int sig) {
  printf("stopping\n");

  //Funzione stacca programma da interfaccia
  //se descrittore programma è -1;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  exit(0);
}

int main(int argc, char **argv) {

//Passato nome del programma eBPF da caricare;
  char *filename="firewall_10240_kern.o";
  __u32 key;
int result;


   // Carica il file oggetto;
```

```
        if (load_bpf_file(filename)) {
    printf("error %s", bpf_log_buf);
    return 1;
  }

  if (!prog_fd[0]) {
    printf("load_bpf_file: %s\n", strerror(errno));
    return 1;
  }

  //Premendo CTRL-C esegue funzione "int_exit" che esce da programma;
  signal(SIGINT, int_exit);

   //Termina programma;
  signal(SIGTERM, int_exit);


  //map_fd[0] ---> riferita alla prima mappa defina nel kern (port_map);

  int q;

  //Ciclo per assegnare 10239 numeri di porta,  superiori a 10000,
  //alle chiavi, e i rispettivi valori;
  for(q=10000; q<10000+10240-1; q++) {
  __u16 port_value = XDP_PASS;
  key = (__u32) q;
  result = bpf_map_update_elem(map_fd[0], &key, &port_value, BPF_ANY);

   //Se restituito un errore;
  if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n",
            errno, strerror(errno));
    return 1;
  }
  }

  //Aggiungo ultima chiave che ha volere diverso dalle altre;
  key = (__u32) 9999;
  __u16 port_value = XDP_DROP;
    result = bpf_map_update_elem(map_fd[0], &key, &port_value, BPF_ANY);
    if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n", errno, strerro
r(errno));
    return 1;
  }

  //Carica programma xdp sull'interfaccia;
  if (bpf_set_link_xdp_fd(ifindex, prog_fd[0], xdp_flags) < 0) {
    printf("link set xdp fd failed\n");
    return 1;
```

```
  }

  int i, j;
  __u32 key2 = 0;

  //Memorizza numero di core a disposizione;
  unsigned int nr_cpus = bpf_num_possible_cpus();

  //Creo array grande quanto numero di core;
  __u64 values[nr_cpus];

  /*Ciclo che stampa per 1000 volte il numero
  * di pacchetti scartati da ciascun core;
  */
  for (i=0; i< 1000; i++) {
    assert(bpf_map_lookup_elem(map_fd[1], &key2, values) == 0);
    printf("%d\n", i);
    for (j=0; j < nr_cpus; j++) {
      printf("cpu %d, value = %llu\n", j, values[j]);
    }
    printf("\n\n");
    sleep(2);
  }

  printf("end\n");
  //Terminato il ciclo stacca il programma xdp;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  return 0;

}
```

firewall_10240_kern.c :

```
#include <uapi/linux/bpf.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/if_vlan.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/in.h>
#include <uapi/linux/tcp.h>
#include <uapi/linux/udp.h>
#include "bpf_helpers.h"

//Numeri IP nodi 0 e 2;

#define IP0 3232235521
#define IP2 3232235777
```

```
struct bpf_map_def SEC("maps") port_map = {
        .type        = BPF_MAP_TYPE_HASH,

        .key_size    = sizeof(__u32),
        .value_size  = sizeof(__u16),
        .max_entries = 10240,
};

//Usata per contare quanti pacchetti sono filtrati per core;
struct bpf_map_def SEC("maps") counter_map = {
        .type        = BPF_MAP_TYPE_PERCPU_ARRAY,
        .key_size    = sizeof(__u32),
        .value_size  = sizeof(__u64),
        .max_entries = 1,

};

SEC("xdp_port_filter")
int _xdp_port_filter(struct xdp_md *ctx) {

  //Assegnerò valore porta del pacchetto in arrivo;
  u32 key;

  //Definiti in lato user, ma potranno essere 1 / 2;
  u16 *port_value;

  //Assegnare a "key" il valore della porta del pacchetto in ingresso;
   void *data_end = (void *)(long)ctx->data_end;
   void *data = (void *)(long)ctx->data;

  //Variabili per indirizzi IP;
   u32 ip_src;
   u32 ip_dest;

//Analisi struttura ethernet:
        struct ethhdr *eth = data;
        if (eth + 1 > data_end) {
                return XDP_ABORTED;
        }
        if(ntohs(eth->h_proto) == ETH_P_IP) {

//Analisi struttura IP:
         struct iphdr *iph = data + sizeof(struct ethhdr);
         if (iph + 1 > data_end) {
                 return XDP_ABORTED;
         }

    /*__constant_ntohl ---> per adattare un unsigned long int,
    * assegnandolo da paccetto ad una variabile con il formato giusto;
        */
```

```
        ip_src = __constant_ntohl(iph->saddr);
        ip_dest = __constant_ntohl(iph->daddr);

        //Effettua controllo porte solo per traffico da nodo 0 a 2;
        if ( ip_src == (__u32)IP0 && ip_dest == (__u32)IP2 ) {

         struct tcphdr *tcp = data + sizeof(struct ethhdr) + sizeof(st
ruct iphdr);
         if(tcp + 1 > data_end) {
                 return XDP_ABORTED;
         }
         if((iph->protocol) == IPPROTO_TCP) {

         //__constant_ntohs ---> per adattare un unsigned short int;
                 key = __constant_ntohs(tcp->dest);
                 goto out;
         }

         struct udphdr *udp = data + sizeof(struct ethhdr) + sizeof(st
ruct iphdr);
         if(udp + 1 > data_end) {
                 return XDP_ABORTED;
         }
         if((iph->protocol) == IPPROTO_UDP) {
                 key = __constant_ntohs(udp->dest);
                 goto out;
         }else{
                goto out2;
         }
        }
        return XDP_PASS;
}
return XDP_PASS;

out:

/*Prendiamo il valore 1 / 2 corrispondente alla porta,  caricato nella
* parte user, nella mappa con la corrispondente chiave;
*/
  port_value = bpf_map_lookup_elem(&port_map, &key);
 if(!port_value) {
         return XDP_PASS;
 }

 //Se il valore non è presente nella mappa (== 0) passa il pacchetto;
  if (*port_value == XDP_PASS){
    return XDP_PASS;
  }
  /*Se valore è 1 allora viene scartato il pacchetto
   *e aggiornata la mappa che conta i pacchetti;
```

```
   */
  if (*port_value == XDP_DROP) {

    u64 *filtered_count;
    u64 *counter;
    u32 key2 = 0;
    counter = bpf_map_lookup_elem(&counter_map, &key2);
    if (counter) {
      //Incremento conteggio di pacchetti filtrati;
      *counter += 1;
    }
    return XDP_DROP;
  }

//Se port_value == 2;
out2:
  return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

APPENDICE F

Vengono riportati i file che permettono l'implementazione di un firewall, il quale utilizza come chiavi della mappa delle strutture, essendo così capace di analizzare oltre al numero di porta, come nell'appendice precedente, anche gli indirizzi IP sorgente e destinazione. Ciò permette di velocizzare l'ispezione del traffico dati, poiché ogni istruzione è verificata tramite la funzione di *lookup*.

L'esempio fa riferimento a un firewall con 10240 istruzioni.

firewall_ip_port_kern.c:

```
#include <uapi/linux/bpf.h>
#include <uapi/linux/if_ether.h>
#include <uapi/linux/if_packet.h>
#include <uapi/linux/if_vlan.h>
#include <uapi/linux/ip.h>
#include <uapi/linux/in.h>
#include <uapi/linux/tcp.h>
#include <uapi/linux/udp.h>
#include "bpf_helpers.h"

//Struttura contenente i campi delle chiavi;
struct bpf_flow_keys {
        __u32 src;
        __u32 dst;
        __u32 ports;
};

struct bpf_map_def SEC("maps") ip_port_map = {
        .type        = BPF_MAP_TYPE_HASH,

        .key_size    = sizeof(struct bpf_flow_keys),
        .value_size  = sizeof(__u16),
        .max_entries = 10240,
};

//Usata per contare quanti pacchetti sono filtrati per core;
struct bpf_map_def SEC("maps") counter_map = {
        .type        = BPF_MAP_TYPE_PERCPU_ARRAY,
        .key_size    = sizeof(__u32),
```

```
        .value_size  = sizeof(__u64),
        .max_entries = 1,

};

SEC("xdp_ip_port_filter")
int _xdp_ip_port_filter(struct xdp_md *ctx) {

  /*Assegnerò valore porta,
  * IP sorgente, IP destinazione
  * del pacchetto in arrivo; */
  struct bpf_flow_keys key;

  //Definiti in lato user, ma potranno essere 1 / 2;
    u16 *port_value;

//Assegnare a "key" il valore della porta
// del pacchetto in ingresso;
   void *data_end = (void *)(long)ctx->data_end;
   void *data = (void *)(long)ctx->data;

//Analisi struttura ethernet:
        struct ethhdr *eth = data;
        if (eth + 1 > data_end) {
                return XDP_ABORTED;
        }
        if(ntohs(eth->h_proto) == ETH_P_IP) {

//Analisi struttura IP:
         struct iphdr *iph = data + sizeof(struct ethhdr);
         if (iph + 1 > data_end) {
                 return XDP_ABORTED;
         }

      /*__constant_ntohl ---> per adattare un unsigned long int,
       * assegnandolo da paccetto ad una variabile
       * con il formato giusto;
       */
        key.src = __constant_ntohl(iph->saddr);
        key.dst = __constant_ntohl(iph->daddr);

         struct tcphdr *tcp = data + sizeof(struct ethhdr) + size
of(struct iphdr);
         if(tcp + 1 > data_end) {
                 return XDP_ABORTED;
```

```
        }
        if((iph->protocol) == IPPROTO_TCP) {

      //__constant_ntohs ---> per adattare un unsigned short int;
                key.ports = __constant_ntohs(tcp->dest);
                goto out;
        }

        struct udphdr *udp = data + sizeof(struct ethhdr) + size
of(struct iphdr);
        if(udp + 1 > data_end) {
                return XDP_ABORTED;
        }
        if((iph->protocol) == IPPROTO_UDP) {
                key.ports = __constant_ntohs(udp->dest);
                goto out;
        }else{
               goto out2;
        }
       return XDP_PASS;
}
return XDP_PASS;

out:

/*Prendiamo il valore 1 / 2 corrispondente alla porta,  caricato
nella parte user,
*nella mappa con la corrispondente chiave;
*/
  port_value = bpf_map_lookup_elem(&ip_port_map, &key);
 if(!port_value) {
        return XDP_PASS;
 }

 //Se il valore non è presente nella mappa (== 0) passa il pacche
tto;
  if (*port_value == XDP_PASS){
    return XDP_PASS;
  }
  /*Se valore è 1 allora viene scartato il pacchetto
   *e aggiornata la cmappa che conta;
   */
  if (*port_value == XDP_DROP) {

    u64 *filtered_count;
```

```
    u64 *counter;
    u32 key2 = 0;
    counter = bpf_map_lookup_elem(&counter_map, &key2);
    if (counter) {
      //Incremento conteggio di pacchetti filtrati;
      *counter += 1;
    }
    return XDP_DROP;
  }

//Se port_value == 2;
out2:
  return XDP_PASS;
}

char _license[] SEC("license") = "GPL";
```

firewall_ip_port_user.c:

```
#include <linux/bpf.h>
#include <linux/if_link.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <arpa/inet.h>
#include <netinet/ether.h>
#include <unistd.h>
#include <time.h>
#include "bpf_load.h"
#include <bpf/bpf.h>
#include "bpf_util.h"

// Assegna l'indice dell'interfaccia ens1f1;
static int ifindex = 5;
static __u32 xdp_flags = 0;

//Funzione chiamata per interrompere il file kern;
static void int_exit(int sig) {
  printf("stopping\n");

//Funzione stacca programma da interfaccia
```

```
//se il descrittore programma è -1;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  exit(0);
}

#define IP0 3232235521
#define IP2 3232235777

//Struttura contenente i campi delle chiavi;
struct bpf_flow_keys {
  __u32 src;
  __u32 dst;
  __u32 ports;

};

int main(int argc, char **argv) {

//Passato nome del programma eBPF da caricare;
//char *filename="firewall_ip_port_kern.o";

struct bpf_flow_keys key;

int result;

  // Cambio dei limiti
  struct rlimit r = {RLIM_INFINITY, RLIM_INFINITY};
  if (setrlimit(RLIMIT_MEMLOCK, &r)) {
    perror("setrlimit(RLIMIT_MEMLOCK, RLIM_INFINITY)");
    return 1;
  }

  // Carica il file oggetto;
        if (load_bpf_file(filename)) {
    printf("error %s", bpf_log_buf);
    return 1;
  }

  if (!prog_fd[0]) {
    printf("load_bpf_file: %s\n", strerror(errno));
    return 1;
  }

//Premendo CTRL-C esegue funzione "int_exit"
//che esce da programma;
```

```
  signal(SIGINT, int_exit);

   //Termina programma;
  signal(SIGTERM, int_exit);


  //map_fd[0] ---> riferita alla prima mappa
  // defina nel kern (port_map);

  int q;

 //Ciclo per assegnare 10239 numeri di porta,  superiori a 10000
 //alle chiavi, e i rispettivi valori;
  for(q=10000; q<10000+10240-1; q++) {
  __u16 port_value = XDP_PASS;

  key.src = (__u32)IP0;
  key.dst = (__u32)IP2;
  key.ports = (__u32)q;

  result = bpf_map_update_elem(map_fd[0], &key, &port_value, BPF_
ANY);

   //Se restituito un errore;
  if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n", errno,
strerror(errno));
    return 1;
  }
  }

  //Aggiungo ultima chiave che ha volere diverso dalle altre;
  key.src = (__u32)IP0;
  key.dst = (__u32)IP2;
  key.ports = (__u32) 9999;
  __u16 port_value = XDP_DROP;
    result = bpf_map_update_elem(map_fd[0], &key, &port_value, BP
F_ANY);
    if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n", errno,
strerror(errno));
    return 1;
  }

  //Carica programma xdp sull'interfaccia;
```

```
  if (bpf_set_link_xdp_fd(ifindex, prog_fd[0], xdp_flags) < 0) {
    printf("link set xdp fd failed\n");
    return 1;
  }

  int i, j;
  __u32 key2 = 0;

  //Memorizza numero di core a disposizione;
  unsigned int nr_cpus = bpf_num_possible_cpus();

  //Creo array grande quanto numero di core;
  __u64 values[nr_cpus];

  /*Ciclo che stampa per 1000 volte il numero
  * di pacchetti scartati da ciascun core;
  */
  for (i=0; i< 1000; i++) {
    assert(bpf_map_lookup_elem(map_fd[1], &key2, values) == 0);
    printf("%d\n", i);
    for (j=0; j < nr_cpus; j++) {
      printf("cpu %d, value = %llu\n", j, values[j]);
    }
    printf("\n\n");
    sleep(2);
  }

  printf("end\n");
  //Terminato il ciclo stacca il programma xdp;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  return 0;

}
```

APPENDICE G

In quest'ultima appendice è riportato il codice di riferimento all'ultimo sottocapitolo del capitolo 5, con il quale al variare del numero di ingressi della mappa è stato possibile ottenere una stima delle prestazioni del firewall fino a 42 milioni di istruzioni.

É stato riportato solo il file "_user.c", poiché il codice della parte kernel è uguale a quello dell'appendice D, dal quale è stato cambiato solamente il numero d'ingressi della mappa hash.

firewall2_ip_port_user.c:

```
#include <linux/bpf.h>
#include <linux/if_link.h>
#include <assert.h>
#include <errno.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/resource.h>
#include <arpa/inet.h>
#include <netinet/ether.h>
#include <unistd.h>
#include <time.h>
#include "bpf_load.h"
#include <bpf/bpf.h>
#include "bpf_util.h"

// Assegna l'indice dell'interfaccia ens1f1;
static int ifindex = 5;
static __u32 xdp_flags = 0;

//Funzione chiamata per interrompere il file kern;
static void int_exit(int sig) {
  printf("stopping\n");

  //Funzione stacca programma da interfaccia se descrittore progra
mma è -1;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  exit(0);
```

```
}

#define IP0 3232235521
#define IP2 3232235777

//Struttura contenente le chiavi;
struct bpf_flow_keys {
        __u32 src;
        __u32 dst;
        __u32 ports;
};

int main(int argc, char **argv) {

//Passato nome del programma eBPF da caricare;
char *filename="firewall2_ip_port_kern.o";

struct bpf_flow_keys key;

int result;

  // Cambio dei limiti
  struct rlimit r = {RLIM_INFINITY, RLIM_INFINITY};
   if (setrlimit(RLIMIT_MEMLOCK, &r)) {
    perror("setrlimit(RLIMIT_MEMLOCK, RLIM_INFINITY)");
    return 1;
  }

  // Carica il file oggetto;
        if (load_bpf_file(filename)) {
    printf("error %s", bpf_log_buf);
    return 1;
  }

  if (!prog_fd[0]) {
    printf("load_bpf_file: %s\n", strerror(errno));
    return 1;
  }

  //Premendo CTRL-
C esegue funzione "int_exit" che esce da programma;
  signal(SIGINT, int_exit);

   //Termina programma;
  signal(SIGTERM, int_exit);
```

```
  //map_fd[0] ---
> riferita alla prima mappa defina nel kern (port_map);

  int q;
  //Primo indirizzo ip, coincide con quello del nodo 0;
  unsigned int ip;

  //Per ogni indirizzo ip, dal 192.168.0.1
  //vengono assegnate 55535 porte, per un totale di (763-
1)*55535 chiavi.
  for(ip=3232235521; ip <3232235521+763-1; ip++){
  for(q=10000; q<10000+55536-1; q++) {
  __u16 port_value = XDP_PASS;

  key.src = (__u32)ip;
  key.dst = (__u32)IP2;
  key.ports = (__u32)q;

  result = bpf_map_update_elem(map_fd[0], &key, &port_value, BPF_A
NY);

   //Se restituito un errore;
  if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n", errno, s
trerror(errno));
    return 1;
  }
  }
  }
  //Aggiungo ultima chiave che ha volere diverso dalle altre;
  key.src = (__u32)IP0;
   key.dst = (__u32)IP2;
  key.ports = (__u32) 9999;
  __u16 port_value = XDP_DROP;
    result = bpf_map_update_elem(map_fd[0], &key, &port_value, BPF
_ANY);
    if (result != 0) {
    fprintf(stderr, "bpf_map_update_elem error %d %s \n", errno, s
trerror(errno));
    return 1;
  }

  //Carica programma xdp sull'interfaccia;
  if (bpf_set_link_xdp_fd(ifindex, prog_fd[0], xdp_flags) < 0) {
```

```
    printf("link set xdp fd failed\n");
    return 1;
  }

  int i, j;
  __u32 key2 = 0;

  //Memorizza numero di core a disposizione;
  unsigned int nr_cpus = bpf_num_possible_cpus();

  //Creo array grande quanto numero di core;
  __u64 values[nr_cpus];

  /*Ciclo che stampa per 1000 volte il numero
  * di pacchetti scartati da ciascun core;
  */
  for (i=0; i< 1000; i++) {
    assert(bpf_map_lookup_elem(map_fd[1], &key2, values) == 0);
    printf("%d\n", i);
    for (j=0; j < nr_cpus; j++) {
      printf("cpu %d, value = %llu\n", j, values[j]);
    }
    printf("\n\n");
    sleep(2);
  }

  printf("end\n");
  //Terminato il ciclo stacca il programma xdp;
  bpf_set_link_xdp_fd(ifindex, -1, xdp_flags);
  return 0;

}
```

Bibliografia e Sitografia

[1] «The eXpress Data Path,» [Online]. Available: https://blogs.igalia.com/dpino/2019/01/10/the-express-data-path/. [Consultato il giorno 31 01 2020].

[2] «cilium,» [Online]. Available: https://github.com/cilium/cilium. [Consultato il giorno 31 01 2020].

[3] «Linux Extended BPF (eBPF) Tracing Tools.,» [Online]. Available: http://www.brendangregg.com/ebpf.html. [Consultato il giorno 31 01 2020].

[4] «Introduction to eBPF and XDP,» [Online]. Available: https://www.slideshare.net/lcplcp1/introduction-to-ebpf-and-xdp. [Consultato il giorno 31 01 2020].

[5] «Why is the kernel community replacing iptables with BPF?,» [Online]. Available: https://cilium.io/blog/2018/04/17/why-is-the-kernel-community-replacing-iptables/. [Consultato il giorno 02 02 2020].

[6] «FLATICON,» [Online]. Available: https://www.flaticon.es/icono-gratis/cortafuegos_811683. [Consultato il giorno 11 02 2020].

[7] «QuidsUp - Tutorials,» [Online]. Available: https://quidsup.net/tutorials/. [Consultato il giorno 11 02 2020].

[8] «XDP tutorial,» [Online]. Available: https://github.com/xdp-project/xdp-tutorial. [Consultato il giorno 13 02 2020].

[9] «Guide, BPF and XDP Reference,» [Online]. Available: https://cilium.readthedocs.io/en/v1.3/bpf/. [Consultato il giorno 13 02 2020].

[10] «eBPF and XDP for Processing Packets at Bare-metal Speed,» [Online]. Available: https://sematext.com/blog/ebpf-and-xdp-for-processing-packets-at-bare-metal-speed/. [Consultato il giorno 13 02 2020].

[11] «How to build a kernel with XDP support,» [Online]. Available: https://blogs.igalia.com/dpino/2019/01/02/build-a-kernel/. [Consultato il giorno 13 02 2020].

[12] «eBPF, part 1: Past, Present, and Future,» [Online]. Available: https://ferrisellis.com/content/ebpf_past_present_future/. [Consultato il giorno 13 02 2020].

[13] «A brief introduction to XDP and eBPF,» [Online]. Available:

https://blogs.igalia.com/dpino/2019/01/07/introduction-to-xdp-and-ebpf/. [Consultato il giorno 13 02 2020].

[14] «BPF: A Tour Of Program Types,» [Online]. Available: https://www.linux.com/news/bpf-tour-program-types/. [Consultato il giorno 13 02 2020].

[15] «BPF(2),» [Online]. Available: http://man7.org/linux/man-pages/man2/bpf.2.html. [Consultato il giorno 13 02 2020].

[16] «Introduction to eBPF and XDP,» [Online]. Available: https://mcorbin.fr/pages/xdp-introduction/. [Consultato il giorno 13 02 2020].

[17] «Using eXpress Data Path (XDP) maps in RHEL 8: Part 2,» [Online]. Available: https://developers.redhat.com/blog/2018/12/17/using-xdp-maps-rhel8/. [Consultato il giorno 13 02 2020].

[18] «setsockopt,» [Online]. Available: https://pubs.opengroup.org/onlinepubs/009695399/functions/setsockopt.html. [Consultato il giorno 13 02 2020].

[19] «IP sets,» [Online]. Available: http://ipset.netfilter.org/. [Consultato il giorno 13 02 2020].

[20] «FRnOG,» [Online]. Available: http://www.frnog.org/. [Consultato il giorno 13 02 2020].

[21] «Setup dependencies,» [Online]. Available: https://github.com/xdp-project/xdp-tutorial/blob/master/setup_dependencies.org. [Consultato il giorno 13 02 2020].

[22] «linux/samples/bpf/,» [Online]. Available: https://github.com/torvalds/linux/tree/v5.4/samples/bpf. [Consultato il giorno 13 02 2020].

[23] «eBPF, part 2: Syscall and Map Types,» [Online]. Available: https://ferrisellis.com/content/ebpf_syscall_and_maps/. [Consultato il giorno 13 02 2020].

[24] «11 Hardware,» [Online]. Available: https://docs.cloudlab.us/hardware.html. [Consultato il giorno 13 02 2020].

[25] «CloudLab,» [Online]. Available: https://cloudlab.us/. [Consultato il giorno 13 02 2020].

[26] J.-H. Y. a. J. W.-K. H. Nguyen Van Tu, «Building Hybrid Virtual Network Functions with eXpress Data Path,» 2019. [Online]. Available: http://dl.ifip.org/db/conf/cnsm/cnsm2019/1570564079.pdf. [Consultato il giorno 13 02 2020].

[27] D. G. CHOUDHURY, «XDP-Programmable Data Path in the Linux Kernel,» 2018. [Online]. Available: https://pdfs.semanticscholar.org/1a27/441586bc685d72a40515bbc0ad3b034748e4.pdf. [Consultato il giorno 13 02 2020].

[28] N. V. Joyce Mwangama, «Accelerated Virtual Switching Support of 5G NFV-based Mobile Networks,» 2017. [Online]. Available: https://ieeexplore-ieee-org.ezproxy.unibo.it/stamp/stamp.jsp?tp=&arnumber=8292535. [Consultato il giorno 13 02 2020].

[29] T. H.-J. e. al., «The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel,» 2018. [Online]. Available: https://dl.acm.org/doi/pdf/10.1145/3281411.3281443?download=true. [Consultato il giorno 13 02 2020].