ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

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

Realizzazione di firewall in Linux tramite Express Data Path (XDP)

Elaborato in Laboratorio di Reti e Programmazione di Dispositivi Mobili

Relatore
Prof. Cerroni Walter

Presentata da
Baldù Giuseppe

Anno Accademico 2018/2019

Indice

Introduzione 2

BPF e XDP 4

2.1 Nascita del BPF 4
2.2 Struttura 5
2.2.1 Network Tap 5
2.2.2 Packet Filter 6
2.3 eBPF 7
2.4 Architettura 8
2.5 Set di Istruzioni 8
2.6 Funzioni di supporto 9
2.7 Mappe eBPF 10
2.8 Chiamate a coda 11
2.9 Sicurezza e limitazioni 11
2.10 XDP 12

Toolchain per XDP 14

3.1 LLVM 14
3.1.1 Clang 14
3.2 JIT 15
3.3 Iproute2 15

Funzione di rete ed implementazioni 16

4.1 Firewall 16
4.1.1 Packet filter firewall o stateless firewall 16
4.1.2 Stateful firewall o circuit-level gateway 17
4.1.3 Application firewall o proxy firewall 17
4.2 Firewall con Iptables 17
4.3 Firewall con XDP 18
4.3.1 Generazione delle regole 20

Ambiente di sviluppo dei programmi 22

5.1 CloudLab 22
5.2 Macchine Virtuali per lo sviluppo del kernel bypass 24
5.3 Test sulla larghezza di banda 25
5.3.1 Confronto tra XDP, Iptables e VM 26
5.3.2 Differenza tra E100 e Virtio 27
5.3.3 Numero di regole elevato 28

Conclusioni 29

Appendice A 30

Appendice B 32

Bibliografia e Sitografia 33

Capitolo 1

Introduzione

Lo sviluppo dei prodotti nell'industria delle telecomunicazioni ha tradizionalmente seguito standard rigorosi per garantire stabilità, aderenza ai protocolli e qualità. Le funzioni di rete erano quindi realizzate su hardware dedicato e con software di proprietà dei *vendor*. Questo approccio ha funzionato correttamente nel passato ma esso portava inevitabilmente a lunghi cicli di vita del prodotto e alla dipendenza da harware specifico. L'evoluzione tecnologica e l'aumento della competizione nei servizi di comunicazione ha portato i *service provider* a cercare nuove strade per rivoluzionare lo status quo.

Contemporaneamente l'aumento delle prestazioni dell'harware commerciale ha reso possibile un crescente utilizzo di funzioni di rete virtualizzate o NFV *(Network Function Virtualization)* e quindi la possibilità di realizzare le funzioni di rete direttamente nel software. La virtualizzazione delle reti ha spinto ad una maggiore flessibilità nell'infrastruttura, essendo possibile eseguire le funzioni direttamente su macchine virtuali (*virtual machine*) o su *container.* Tali funzioni possono essere perciò aggiunte, rimosse o modificate in base alle richieste del sistema senza dover realizzare modifiche all'hardware.

A differenza degli altri sistemi operativi, Linux è nato, secondo il volere del fondatore Linus Torvald, con una licenza GPL (General Public License) la quale lo rende un sistema operativo *open source* ed in grado di garantire quattro fondamentali libertà: esecuzione, studio, ridistribuzione e libertà di migliorare il software. Grazie a questa principale caratteristica Linux è il sistema operativo usato in prevalenza sui diversi dispositivi per la realizzazione delle funzioni di rete. La grande diffusione in questo settore è data dalla possibilità di accedere a certe funzionalità che nei sistemi operativi privatizzati non sono accessibili da parte dell'utente.

A discapito della maggior flessibilità negli hardware general purpose le prestazioni difficilmente raggiungono gli hardware dedicati, perciò la ricerca nel settore delle telecomunicazioni sta puntando sempre di più sull'aumento delle prestazioni dei software per le NFV. L'utilizzo di tale architettura richiede che l'elaborazione dei pacchetti e delle funzioni di rete vengano eseguite direttamente nello spazio utente. Questa caratteristica pone alcuni vincoli sulle prestazioni per effetto della necessità di generare una copia dei pacchetti, presenti sul driver della scheda di rete, nell'User Space. Per andare incontro a questa problematica e per aumentare le prestazioni, nelle ultime versioni del kernel Linux sono state integrate alcune funzionalità che permettono di processare i pacchetti in un livello più basso dello stack di rete e quindi eseguire i programmi direttamente nel kernel.

Questo lavoro di tesi pone l'attenzione sulla struttura del Berkeley Packet Filter e sulla sua recente "versione estesa" eBPF. Nel Capitolo 2 viene presentata l'architettura della macchina virtuale BPF e come essa realizza il processamento dei pacchetti direttamente all'interno del

kernel Linux, mostrando le potenzialità e innovazioni dell'eBPF. In particolar modo si vuole effettuare lo studio della tecnologia eXpress Data Path (XDP) per eBPF e delle sue prestazioni. Essa introduce alcuni *hook* nel kernel che riescono ad "agganciare" i pacchetti direttamente dal driver di rete. Per andare a testare le effettive prestazioni dell'XDP si è voluto realizzare un programma eBPF che implementa la funzione di rete virtualizzata di un firewall. Nel Capitolo 3 viene effettuata una panoramica sull'ambiente di sviluppo per questa tipologia di programmi andando a descrivere tutti i *tool* necessari. Nel Capitolo successivo verrà svolta una panoramica inerente all'evoluzione dei firewall e verranno presentati i diversi programmi realizzati. I programmi andranno a generare la stessa funzione di rete con le medesime caratteristiche su quattro diversi ambienti, in modo tale da poter analizzare il confronto tra l'analisi interna al kernel e le tecniche di *kernel bypass*. Nel Capitolo 5 verranno presentati i risultati dei test effettuati ponendo a confronto il programma XDP con quello che è attualmente il metodo più comune per la realizzazione dei firewall di rete in ambiente Linux, Iptables. Verrà effettuato il confronto con una macchina virtuale (E1000) che, a differenza dell'eBPF, lavora nello spazio utente ed effettua la "virtualizzazione completa" dell'ambiente di lavoro. Si è voluto inoltre testare le prestazioni di una macchina virtuale Virtio che sfrutta la paravirtualizzazione e verificare le differenze tra le due diverse tipologie di VM.

Capitolo 2

BPF e XDP

In questo capitolo si vuole presentare la struttura della macchina virtuale BPF sia dal punto di vista del funzionamento che dell'architettura. Inoltre si andrà ad approfondire la nuova versione del BPF, denominata eBPF (extended BPF) con particolare attenzione alle differenze che sono state introdotte e alle nuove potenzialità che offre. Tra le principali verranno descritte le funzioni di supporto,le chiamate a coda e le mappe eBPF. Infine verrà descritto il tipo di programma XDP per eBPF che sarà oggetto di analisi per l'implementazione di un firewall di rete.

2.1 Nascita del BPF

I meccanismi per la cattura dei pacchetti da parte delle applicazioni di norma vengono svolti nello spazio utente. Di conseguenza anche il monitoraggio della rete viene svolto all'esterno del kernel. Questo metodo però richiede un meccanismo che crei una copia dei pacchetti e la trasferisca dal kernel allo spazio utente. La copia viene realizzata inserendo un *agent* nel kernel chiamato *packet filter*. Uno dei primi creati fu il CMU/Stanford Packet Filter (detto anche CSPF) realizzato nel 1980. Questa prima idea di agent fornì una struttura necessaria ed ampiamente utilizzata la quale ha spianato la strada verso nuovi miglioramenti e nuove soluzioni. McCanne e Jacobson nel loro paper pubblicato nel 1992 dal titolo "The BSD Packet Filter: A New Architecture for User-level Packet Capture" [1] affermano però quanto citato in riferimento all'obsoleto CSPF:

"However, on today's machines its performance, and the performance of its descendents, leave much to be desired".

Il BSD Packet Filter (BPF) fu creato da Steven McCanne & Van Jacobson ai Lawrence Berkeley Laboratory nel 1992 e presentato ufficialmente nel gennaio del 1993 alla Winter USERNIX conference a San Diego (CA). Esso consiste in una macchina virtuale per dispositivi basati su Unix, nato per eseguire il processing dei pacchetti non più nell'user-space ma direttamente nel kernel. Grazie a questa sua peculiarità risulta possibile realizzare l'analisi e il filtraggio dei pacchetti nel livello Datalink (livello 2) della pila OSI. Tale caratteristica permette di eseguire il processamento dei pacchetti nel modo più efficiente e sarà possibile trasferire nell'User-space solo i pacchetti da analizzare, riuscendo ad avere notevoli guadagni in termini di performance.

Uno dei principali esempi di applicazioni Linux, tutt'ora in uso, che utilizza il BPF è il Tcpdump, per il monitoraggio dei pacchetti in transito su un'interfaccia di rete.

BPF a differenza del CSPF attua delle scelte strutturali che lo fanno essere migliore. In particolar modo BPF sfrutta i registri della macchina a differenza del CSPF, che si basava su uno stack di memoria, e implementa un modello a buffer non condivisi che si adatta bene alle nuove architetture con un grande spazio di indirizzamento. Come affermano gli autori però il CSPF fu usato come base di partenza per lo sviluppo del BPF.

2.2 Struttura

Lo scopo principale dei programmi BPF consiste nel generare una copia dei pacchetti provenienti dalla rete all'interno del kernel e ridistribuirle alle applicazioni in ascolto nello spazio utente in maniera efficiente. Possiamo suddividere la struttura della macchina virtuale BPF in due componenti: il Network Tap e il Packet Filter.

2.2.1 Network Tap

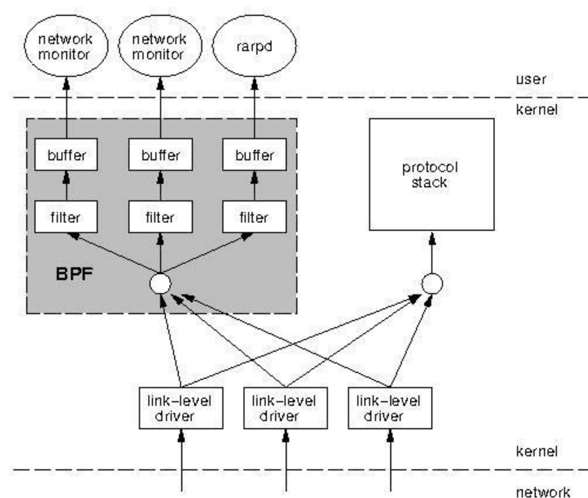

Figura 1: Panoramica sulla struttura del BPF [1]

In assenza del programma BPF, il pacchetto verrebbe portato dal driver direttamente nello stack di protocollo. Se il BPF è attivo il driver fa sì che la prima operazione sia far elaborare il pacchetto dal filtro BPF. Così facendo solo i pacchetti "utili" verranno successivamente salvati in un buffer associato al filtro e consegnati alle applicazioni in ascolto a livello user o allo stack di protocollo quando richiesto. Per velocizzare l'analisi il BPF riempie il buffer con più pacchetti, inserendo però un'header che specifica alle applicazioni che tipo di pacchetti sono presenti e in che posizione del buffer si trovano.

Un notevole vantaggio che ha il BPF, a differenza degli altri metodi, è quello di riuscire ad analizzare i pacchetti ancor prima di generarne una copia. Questa caratteristica velocizza notevolmente il processo, infatti non verrà sprecato tempo, spazio e capacità di calcolo della CPU per salvare in memoria e analizzare pacchetti che poi dovranno essere scartati. Un esempio di Packet Filter che implementa quest'ultima modalità è il NIT in SunOs della Sun, il quale copia sempre il pacchetto in uno *stream buffer* prima di passarlo al filtro.

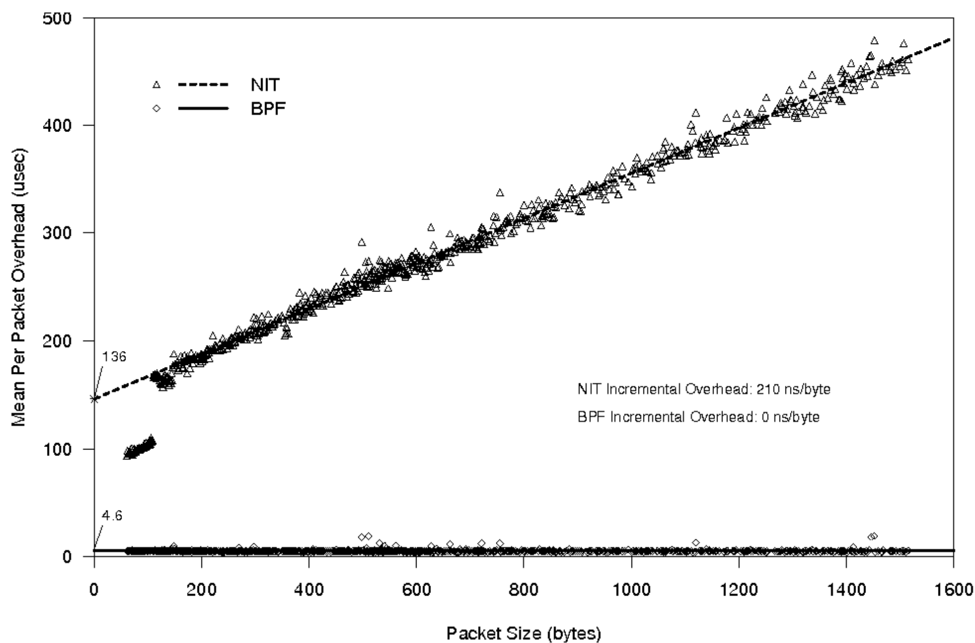

Figura 2: Confronto tra NIT e BPF [1]

In Figura 2 è presentata la differenza in termini di prestazioni tra un filtro BPF e un filtro NIT entrambi posti in modalità *"reject all"* in cui ogni filtro non farà passare nessun pacchetto. L'operazione di DROP generalmente è l'operazione più rapida che si può eseguire e come si può notare dalla figura l'analisi direttamente nel kernel è molto più vantaggiosa di quella nello spazio utente.

2.2.2 Packet Filter

Per il monitoraggio della rete quasi sempre è necessario esaminare solo una parte dei pacchetti in transito, questo fa sì che il filtro debba essere il più performante possibile. Si ha come conseguenza che l'intero rendimento della macchina BPF dipenda quasi esclusivamente dal rendimento del filtro.

Un Packet Filter si può riassumere come una funzione che implementa un semplice controllo booleano sul pacchetto. Se la funzione restituisce *true* il kernel copia il pacchetto nello spazio utente, altrimenti lo ignora. Il BPF utilizza un grafo diretto aciclico di controllo del flusso (CFG). In questo tipo di grafo i nodi rappresentano i vari controlli effettuati sul pacchetto, mentre gli archi, i trasferimenti di controllo. Ogni controllo genera sempre e solo due archi (*true* e *false*).

Il vantaggio della rappresentazione con questa tipologia di grafo è che esso ricorda l'analisi (*parsing*) fatta in precedenza, poichè può ogni volta essere riorganizzato, in modo tale che il valore restituito dalla funzione venga direttamente utilizzato ai nodi alla fine di tutti i controlli. In Figura 3 è presente un esempio di grafo realizzato su un controllo dell'intestazione di un'indirizzo. In questo scenario i protocolli controllati sono solo IP e ARP.

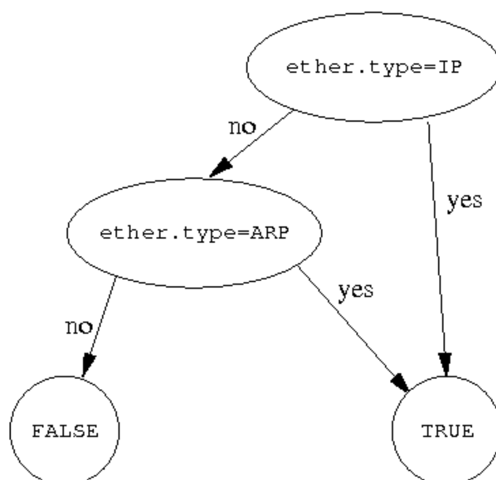

Figura 3: Rappresentazione di un Control Flow Graph [1]

Di seguito vediamo un elenco delle principali caratteristiche che sono state utilizzate per generare i filtri che secondo gli autori sono da considerare come vincoli progettuali per un'evoluzione futura:

- Il BPF deve essere indipendente dal protocollo
- Il BPF deve essere generico
- I riferimenti ai dati nei pacchetti devono essere minimizzati
- I registri della macchina astratta dovrebbero risiedere all'interno dei registri di una macchina reale, essendo i registri fisici molto più performanti

2.3 eBPF

La capacità di eseguire programmi forniti dall'utente all'interno del kernel si è rivelata una decisione di progettazione utile, ma non tutti gli aspetti del BPF sono stati al passo con i tempi. Primo fra tutti l'architettura della macchina virtuale e il set di istruzioni ISA (Instruction Set Architecture). La tecnologia del BPF quindi non corrispondeva più alle realtà dei moderni processori i quali avevano espanso la dimensione dei registri a 64 bit e implementato le CPU multi-core.

Per sfruttare a pieno le nuove potenzialità dell'hardware, Alexei Starovoitov ha introdotto una nuova versione del BPF, che chiameremo Extended-BPF o più semplicemente eBPF. Per differenziarlo dal BSD Packet Filter che fa riferimento alla prima versione di Berkeley Packet Filter presentato nel paragrafo precedente, da ora in poi verrà nominato Classic-BPF (cBPF).

Per quanto riguarda la trasferibilità dei programmi, sulle nuove versioni di Linux vi è una traduzione automatica e trasparente del cBPF in eBPF prima dell'esecuzione del programma, a patto che sia presente un compilatore JIT (argomento che verrà trattato nel capitolo 3 inerente all'ambiente di sviluppo) per eBPF.

2.4 Architettura

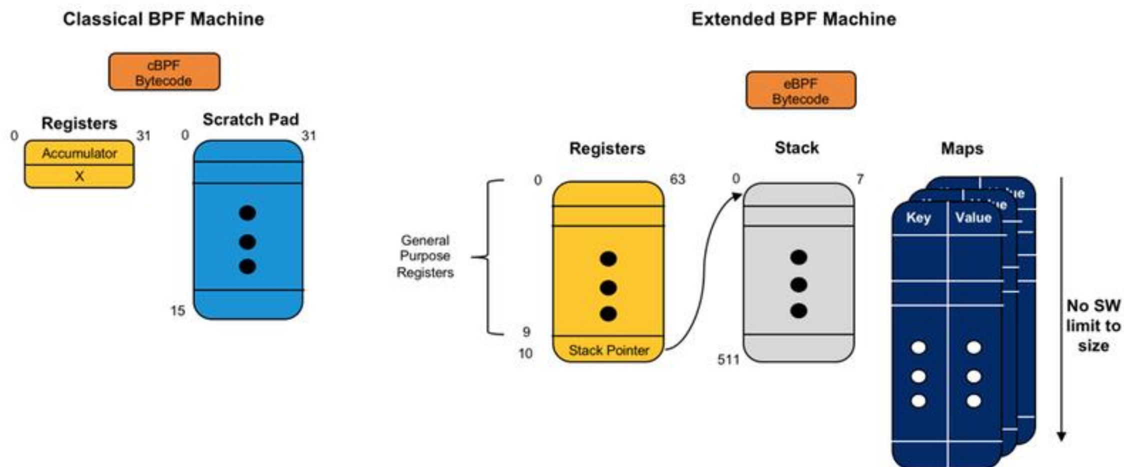

Figura 4: Architetture delle macchine BPF ed eBPF a confronto [3]

Il BPF classico (cBPF) consisteva in un accumulatore e un registro 'X' a 32 bit, in più erano presenti 16 registri a 32 bit che venivano usati come memoria *scratch*. Una delle modifiche più importanti dell'eBPF dal punto di vista dell'architettura è stato il passaggio a 11 registri a 64 bit ciascuno con il supporto di load/store arbitrarie. Grazie a questa nuova architettura è possibile passare i parametri alle funzioni nei registri delle macchine virtuali eBPF, proprio come sull'hardware nativo. Oltre ad aver ampliato i registri sono stati inseriti 512 Byte per lo Stack e un Program Counter. eBPF offre la possibilità di implementare delle mappe, porzioni di memoria che possono essere allocate in base alle necessità e condivise tra User e Kernel space, o addirittura tra più programmi eBPF eseguiti in parallelo, fungendo da archivi chiave/valore efficienti. Tra le altre potenzialità offerte da eBPF c'è la possibilità di implementare le chiamate a coda per programmi eBPF (*tail calls*).

2.5 Set di Istruzioni

L'ISA dell'eBPF è composta da un set di istruzioni di tipo RISC. Attualmente ne sono state implementate 87 con la possibilità di estendere il set a nuove istruzioni se necessario. Le istruzioni possono essere raggruppate in 4 macro categorie: Load, Store, ALU e Jump (condizionati e incondizionati).

eBPF è stato originariamente progettato per scrivre programmi in *C-restricted*, ovvero in una versione che non permette di eseguire tutte le istruzioni *C*. Alcune tra le più comuni restrizioni sono l'impossibilità di implementare cicli *loop* infiniti, non sono consentite variabili globali e non è possibile generare matrici.

Il set di istruzioni è stato originariamente progettato per essere compilato in eBPF attraverso un back-end del compilatore come ad esempio Clang di LLVM che genera un file oggetto, in modo che il kernel possa successivamente mapparli in codici operativi nativi con un compilatore JIT (Just In Time) ed avere così prestazioni ottimali al suo interno.

I vantaggi presentati da questo “ambiente di lavoro” sono:

- Rendere il kernel riprogrammabile senza dover passare dallo spazio utente ogni volta che si necessita di una modifica
- I programmi possono essere aggiornati istantaneamente senza la necessità di dover riaggiornare il kernel e quindi senza interruzioni di traffico
- È possibile mantenere qualsiasi stato del programma anche durante gli aggiornamenti, grazie alle mappe
- Fornisce un’ABI (Application Binary Interface) stabile verso lo spazio utente
- I programmi sono portabili su diverse architetture e fanno uso dell’infrastruttura del kernel esistente

2.6 Funzioni di supporto

Una delle nuove potenzialità che offre eBPF è la possibilità di chiamare delle funzioni di supporto o *helper function* dall'interno di un programma. Queste funzioni sono limitate a una lista di helper definiti nel kernel. Tali funzioni infatti sono parte del kernel principale e non possono essere estese o aggiunte tramite moduli secondari. Al momento sono disponibili 38 diversi helper eBPF.

Questi helper vengono utilizzati dai programmi eBPF per interagire con il sistema o con il contesto in cui lavorano. Ad esempio, possono essere utilizzati per stampare messaggi di debug, per ottenere il tempo dall'avvio del sistema, per interagire con le mappe o per manipolare i pacchetti di rete.

Poiché esistono diversi tipi di programmi eBPF e non vengono eseguiti nello stesso contesto, ciascun tipo di programma può chiamare solo un sottoinsieme di tali helper. Per convenzione ogni helper può avere al massimo 5 argomenti e la chiamata di una funzione di supporto è molto simile ad una chiamata di sistema. Esse vengono invocate mediante 6 nuove istruzioni: da `BPF_CALL_0()` fino a `BPF_CALL_5()`.

Il vantaggio di queste funzioni è la possibilità che hanno i programmi di chiamarle direttamente senza richiedere nessuna interfaccia esterna, essendo le funzioni già compilate. Ogni funzione di supporto verrà compilata in modo trasparente ed efficiente tramite il compilatore JIT. Questo non comporta perdita di prestazioni in termini di sovraccarico, dal momento che le assegnazioni del registro BPF corrispondono già ai registri dell’architettura sottostante.

2.7 Mappe eBPF

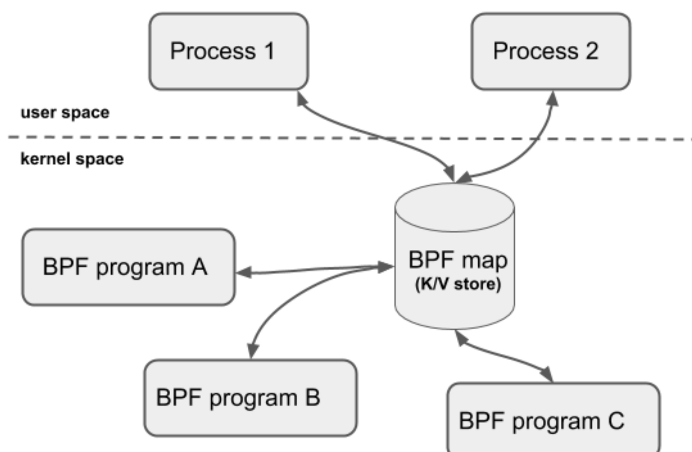

Figura 5: Esempio di possibili connessioni mappe/programmi [4]

Come accennato nel paragrafo 2.4 in cui viene descritta l'Architettura dell'eBPF, le mappe sono efficienti archivi chiave/valore che risiedono nel kernel. È possibile accedervi da un programma eBPF per mantenere lo stato tra più invocazioni di programmi. È inoltre possibile accedervi tramite descrittori di file dallo spazio utente e possono essere arbitrariamente condivisi con altri programmi eBPF o applicazioni dello spazio utente.

Ogni programma può condividere più mappe, anche se un singolo programma attualmente può accedere direttamente solo a 64 diverse. Ogni mappa può essere condivisa con più programmi, con il vincolo che i programmi che condividono mappe tra loro non devono essere dello stesso tipo. Il vantaggio dell'utilizzo delle mappe risiede anche nel fattore memoria: come vedremo nel paragrafo 2.9 il numero di istruzioni di un programma eBPF è limitato a 4096, mentre andando ad utilizzare le mappe è possibile assegnare quantità maggiori di memoria da utilizzare nei diversi programmi.

Esistono diverse tipologie di mappe, ma è possibile raggrupparle in due categorie: Array e Hash. Le mappe di tipo Array allocano direttamente tutte le chiavi numerate in ordine crescente e tutti i valori inizializzandoli a zero. Sarà quindi possibile andare ad assegnare i valori alle rispettive chiavi. Le mappe di tipo Hash inizialmente non contengono nessun elemento, per cui è necessario definire sia la chiave che il valore di ogni elemento in fase di generazione. Di seguito viene presentata una parte della libreria `bpf.h` che contiene alcune delle possibili tipologie di mappe.

```
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,
        …
};
```

2.8 Chiamate a coda

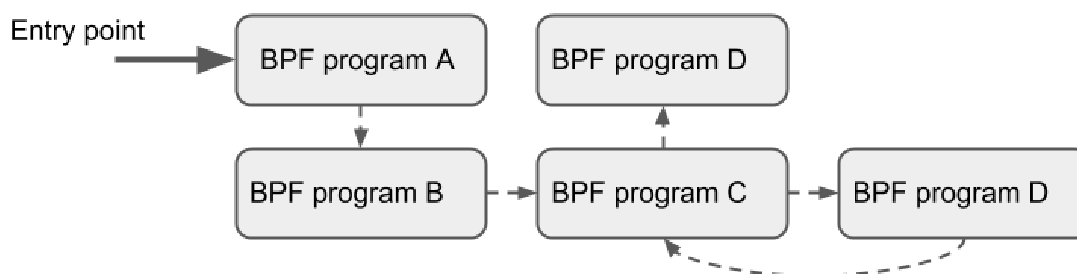

Figura 6: Esempio di flusso di programmi connessi da tail calls [4]

Le chiamate a coda possono essere viste come un meccanismo che consente ad un programma BPF di chiamarne un altro, senza tornare al precedente. Tale chiamata ha un sovraccarico minimo in quanto, a differenza delle chiamate di funzione, è implementata come un *jump*, riutilizzando lo stesso *stack frame.* I programmi sono verificati indipendentemente, perciò sarà necessario utilizzare le mappe per il trasferimento dello stato. Un vincolo che pone questa tipologia di chiamata è dato dal tipo di programmi. Solo i programmi dello stesso tipo possono effettuare le chiamate a coda.

Per eseguire una chiamata a coda è necessario allocare una mappa di tipo `BPF_MAP_TYPE_PROG_ARRAY`, andando ad attribuire i valori rispettivamente ai descrittori dei file della coda. Successivamente sarà necessario utilizzare una funzione di supporto (`bpf_tail_call`) con cui viene passato il contesto.

2.9 Sicurezza e limitazioni

Essendo i programmi eseguiti direttamente nel kernel è necessario garantire un certo livello di sicurezza. Affinchè il programma caricato non vada a compromettere il corretto funzionamento è stato implementato un verificatore eBPF, che andrà ad analizzare il codice. Compito fondamentale del verificatore è il controllo della terminazione del programma:

- Il numero massimo di istruzioni per programma è limitato a 4096
- Sono proibiti loop infiniti, il programma deve sempre terminare
- Le istruzioni di Jump dovranno accedere a porzioni di memoria valide
- È necessario che vi sia una correttezza sintattica delle diverse istruzioni

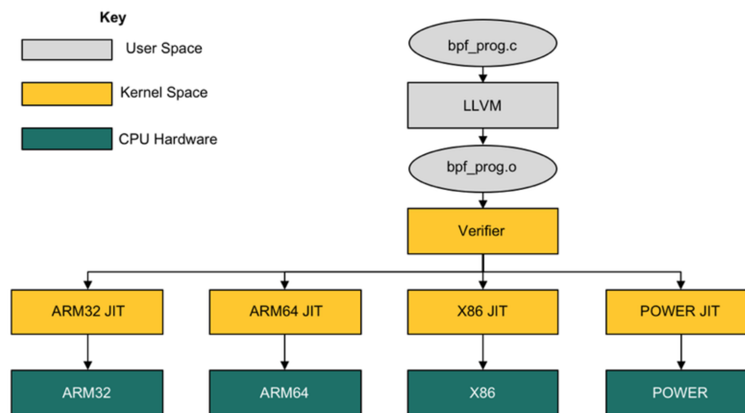

Figura 7: Flusso dicompilazione di un programma eBPF sull'host [3]

Come si può notare in Figura 7 il verificatore è posto tra il compilatore LLVM e il compilatore JIT in modo che possa rifiutare qualsiasi programma non soddisfi le caratteristiche sopra elencate ancor prima di caricare il codice nel kernel.

2.10 XDP

Le tecniche di bypass del kernel per eseguire l'elaborazione di pacchetti programmabili stanno aumentando notevolmente. Queste tecniche risultano però non molto performanti dal momento che tutta la gestione dei pacchetti viene effettuata nello spazio utente. *XDP* o *eXpress Data Path* fornisce, ad alte prestazioni, un percorso dati di rete programmabile all'interno del kernel Linux. XDP consente di elaborare i pacchetti nel punto più basso dello stack del software, mantendo sempre un'integrazione con il resto del sistema e quindi riuscendo a garantire la sicurezza data dal *eBPF verifier*.

Ideale per funzionare ad alte velocità senza comprometterne la programmabilità, XDP è basato su eBPF ed è stato unito per la prima volta nella versione del kernel Linux 4.8. Altri vantaggi chiave che include XDP sono: l'assenza di richiesta sia di un'hardware specializzato, sia del bypass del kernel ed il supporto di qualsiasi tipo di scheda di rete che abbia un driver Linux.

L'idea di base di XDP è quella di aggiungere un *hook* (gancio) iniziale nel percorso del kernel che riceve pacchetti. Successivamente sarà il programma eBPF che andrà ad elaborare il pacchetto, se necessario. L'*hook* viene inserito nel driver NIC subito dopo l'elaborazione dell'interrupt e prima di qualsiasi allocazione di memoria richiesta dallo stack di rete stesso, poiché l'allocazione della memoria, come abbiamo visto per il NIT, è un'operazione costosa. Grazie a questo design, XDP può "far cadere" 26 milioni di pacchetti al secondo per Core con hardware di base, secondo Høiland-Jørgensen [8]. Il programma eBPF è autorizzato a modificare i dati del pacchetto se richiesto, in particolar modo potrà analizzare i dati del pacchetto tramite "accesso diretto". L'eBPF contiene i puntatori dei dati direttamente nei registri e quindi può caricare il contenuto e scrivere direttamente il pacchetto. A seguire un codice-azione definito in XDP effettuerà l'azione stessa sul pacchetto. Attualmente XDP implementa 5 azioni possibili:

- XDP_PASS: il programma decide di passare il pacchetto al normale stack di rete per l’elaborazione
- XDP_DROP: lascia cadere il pacchetto, non esiste metodo più veloce
- XDP_TX: restituisce alla stessa interfaccia di rete il pacchetto, solitamente prima della scrittura viene elaborato. Può implementare il load balancer
- XDP_REDIRECT: restituisce il pacchetto ad un’altra interfaccia di rete
- XDP_ABORTED: provoca il rilascio del pacchetto come XDP_DROP ma restituendo un codice di ritorno al programma eBPF in caso di errore nel programma

I programmi di rete eBPF, in particolar modo per XDP, possono avere un’interfaccia di offload per l’hardware nel kernel in modo da poter eseguire il codice direttamente sulla scheda di rete. Attualmente solo il kernel driver di Netronome ha il supporto per l'offload dei programmi attraverso un compilatore JIT che traduce le istruzioni eBPF in un set di istruzioni implementato rispetto alla scheda NIC.

Capitolo 3

Toolchain per XDP

La macchina virtuale eBPF permette una programmazione ad alto livello utilizzando un linguaggio *C-restricted*. Questa possibilità facilita notevolmente la programmazione, ma necessita di un compilatore per la creazione di un file oggetto. Nel progetto è stato utilizzato il compilatore Clang, facente parte del progetto LLVM, il quale presenta compilazioni molto più rapide di altri. Un altro elemento fondamentale necessario all'eBPF per poter funzionare correttamente è il compilatore JIT che andrà a tradurre le istruzioni in codice nativo. Per lo sviluppo dei programmi XDP in un'ambiente Linux è necessario avere un *front-end* per la configurazione delle interfacce ed il caricamento dei programmi direttamente su di esse. Al fine di caricare il programma del firewall sulla rete è stato utilizzato Iproute2. Nei paragrafi successivi andremo a vedere nello specifico i *tool* sopracitati per la configurazione dell'ambiente di sviluppo di un programma XDP per eBPF.

3.1 LLVM

LLVM è una raccolta di tecnologie di compilazione e *toolchain* modulari e riutilizzabili. L'attività è iniziata come progetto di ricerca dell'Università dell'Illinois, con l'obbiettivo di fornire una moderna strategia di compilazione basata su SSA (*Static Single Assignment*) ed in grado di supportare sia la compilazione statica che quella dinamica di linguaggi arbitrari. Da allora si è sviluppato in diversi sottoprogetti, molti dei quali vengono utilizzati nella produzione di una vasta gamma di progetti open source e nella ricerca accademica.

3.1.1 Clang

Clang è un compilatore C/C++/*Objective*-C che fa parte di uno dei sottoprogetti pricipali di LLVM. Ha come obbiettivo quello di fornire compilazioni sorprendentemente veloci (circa 3 volte GCC). Inoltre è in grado di mostrare un debug in fase di compilazione, generando messaggi di errore.

Clang viene utilizzato nello spazio utente per generare il file oggetto, che successivamente andrà compilato con un compilatore JIT. In Figura 7 si può osservare il flusso di compilazione di un programma eBPF su alcune architetture CPU.

Di seguito un'esempio di istruzione utilizzata per la creazione di un object file `xdp.o` a partire dal file `xdp.c`.

```
clang -O2 -target bpf -c xdp.c -o xdp.o
```

3.2 JIT

I compilatori JIT (Just In Time) permettono un tipo di compilazione effettuata durante l'esecuzione del programma, a differenza degli altri compilatori che la eseguono precedentemente. Il tipo di compilazione effettuata può essere descritta come traduzione dinamica del codice. Lo scopo del compilatore JIT è quello di combinare i vantaggi della compilazione *bytecode* a quelli della compilazione in linguaggio macchina, aumentando le prestazioni quasi alla pari della compilazione direttamente in linguaggio nativo. Infatti in un ambiente JIT la prima fase è costituita dalla compilazione in *bytecode* e solo in fase di esecuzione verrà tradotto in linguaggio nativo. Da qui prende il nome di "Just In Time". L'effetto dell'utilizzo di un compilatore JIT è quello di accellerare notevolmente l'esecuzione del programma riducendo il costo per istruzione. Spesso le istruzioni sono mappate 1:1 con quelle native dell'architettura e quindi si ha come conseguenza una notevole riduzione dell'immagine eseguibile.

Le architetture `x86_64`, `arm64`, `ppc64`, `s390x`, `mips64` e `sparc64` a 64 bit e `x86_32` a 32 bit sono già fornite di un compilatore JIT per eBPF all'interno del kernel.

3.3 Iproute2

Iproute2 è una collezione di *utility* dello spazio utente per il controllo e il monitoraggio di diversi aspetti dell'infrastruttura di rete del Kernel Linux. È un progetto open source che presenta al suo interno la possibilità di allocare sulle diverse NIC i programmi XDP.

Per caricare il file nel kernel è necessario utilizzare il comando `ip link` che permette di settare su quale connessione fisica (porta di rete) disporre il programma XDP.

```
ip link set dev ens1f1 xdp obj xdp.o sec prog
```

In questo esempio è stato caricato il file oggetto `xdp.o` sull'interfaccia fisica `ens1f1`. Il flag `xdp` sta a significare che il kernel caricherà il programma sulla scheda di rete come se fosse "nativo". Il nome della sezione (`prog`) è definito dal programmatore.

In caso si voglia eliminare il file dall'interfaccia, sarà sufficiente utilizzare il comando:

```
ip link set dev ens1f1 xdp off
```

Per il corretto funzionamento è necessario andare a rimuovere il programma prima del caricamento successivo, perché la scheda di rete permette il caricamento di un solo programma per volta.

Capitolo 4

Funzione di rete ed implementazioni

In questo capitolo viene presentata una panoramica sul funzionamento e sulle varie implementazioni della funzione di rete scelta per il test, ovvero quella di firewall. In ambiente Linux attualmente il metodo più utilizzato è attraverso Iptables. Nel paragrafo 4.2 verrà descritto il firewall Linux ed in che modo è possibile effettuare i filtri al suo interno. Successivamente verrà descritto il programma XDP, come possibile alternativa ad Iptables.

4.1 Firewall

In ambito informatico quando si parla di Firewall si fa riferimento ad un filtro software in grado di garantire la sicurezza delle reti. Solitamente viene posto in protezione di una rete privata (o interna) che scambia traffico con la rete pubblica (o esterna). Il filtro è necessario per proteggere la rete interna da attacchi provenienti dall'esterno e per proteggere la rete esterna da un potenziale virus presente sulla rete interna. Tale filtro può essere semplicemente un programma installato su un PC o una macchina dedicata a tale scopo. Solitamente i firewall realizzano una decisione booleana sul controllo che accetta o rifiuta un pacchetto. Nel corso della storia il firewall ha avuto un'evoluzione crescente dal punto di vista della sicurezza. Questo ha reso sempre più complesso il controllo effettuato sul pacchetto. Di seguito vengono presentate in breve le principali tipologie di *Packet Filter* implementate nel corso della storia.

4.1.1 Packet filter firewall o stateless firewall

La più semplice implementazione di Packet Filter è realizzata ponendo il filtro software che analizza il traffico tra la rete pubblica e quella privata. In questa tipologia, definita appunto *stateless* (senza stato), vengono utilizzate solo alcune informazioni presenti nell'header del pacchetto, in particolar modo quelle presenti nei primi 3 livelli della pila OSI ed alcune del livello 4.

Solitamente i filtri vengono realizzati con i seguenti controlli: IP sorgente, IP destinazione, porta sorgente, porta destinazione e protocollo di trasporto. Viene eseguito un controllo semplice e leggero ma non molto sicuro.

Non riuscendo a controllare lo stato della connessione si può facilmente aggirare con una tecnica definita *IP spoofing*, la quale consiste nell'andare a sostituire l'indirizzo IP sorgente di un pacchetto entrante in modo che possa essere accettato dal firewall.

4.1.2 Stateful firewall o circuit-level gateway

Il firewall Statefull è simile al firewall stateless ma riesce a tenere traccia dello stato della connessione, questo comporta che si abbiano regole più complesse. Il controllo dello stato amplia il campo dei controlli e quindi risolve il problema dell'IP spoofing. Entrambe le tipologie di filtri non rilevano attacchi informatici ad un livello superiore al 4, ed in particolar modo questa tipologia è sensibile agli *attacchi DoS* che pongono il sistema in sovraccarico a tal punto che la macchina non riesca più a fornire i servizi richiesti.

4.1.3 Application firewall o proxy firewall

A differenza dei casi precedenti il proxy firewall opera fino al livello 7 del modello OSI. Così facendo, si avrà un'analisi dell'intero pacchetto considerandone anche il contenuto.

Una caratteristica necessaria per il corretto funzionamento consiste nell'avere il firewall come unico punto di accesso con la rete esterna, in modo che tutto il traffico dovrà passare attraverso il filtro. Questo potrebbe generare un effetto indesiderato di collo di bottiglia nella rete che ne potrebbe rallentare notevolmente le prestazioni.

Molti firewall spesso sono associati alle funzioni di NAT (Network Address Translation), possono registrare tutte le operazioni fatte (*logging*) e quindi riuscire a tenere statistiche di quali regole sono state più violate.

4.2 Firewall con Iptables

Iptables fa parte del progetto Netfilter ed è un potente software per la manipolazione dei pacchetti integrato nel kernel Linux. È possibile utilizzarlo direttamente da linea di comando e permette ad un'amministratore di rete di configurare delle tabelle provviste dal firewall nel kernel. Iptables andrà quindi ad effettuare un'analisi dei pacchetti nello stack di rete del kernel. Queste tabelle sono rispettivamente: Raw, Filter, Nat e Mangle. Iptables implementa le funzionalità di un firewall stateful e permette di controllare il traffico in transito su tutte le interfacce di rete (loopback compreso). Ogni tabella a sua volta è composta da gruppi di regole denominati *chain.* Le funzionalità del firewall vere e proprie sono implementate dalla tabella *filter* ed il controllo del traffico è definito dalla *forward chain***.** Ogni chain contiene un elenco di regole disposte secondo l'ordine di inserimento. Ogni regola può stabilire se scartare (DROP), rifiutare esplicitamente (REJECT) o accettare (ACCEPT) un pacchetto. Se un pacchetto non soddisfa nessuna regola verrà applicata la regola o policy di default di quella chian. Iptables essendo una "tabella" dovrà essere scritta in ordine dalla regola più restrittiva a quella più generale, in modo da non accettare pacchetti che poi verrebbero scartati da una regola successiva.

Di seguito sono presentati alcuni dei controlli possibili di Iptables:

- Interfaccia di rete di ingresso o di uscita
- Indirizzo MAC di origine
- Indirizzo IP sorgente e destinazione (dell'host o della rete)

- Protocollo (TCP, UDP, ICMP, …)
- Porta sorgente e destinazione
- Tipo di messaggio ICMP
- Stato della connessione (NEW, ESTABLISHED, INVALID, UNTRACKED, …)
- …

Di seguito viene mostrato un esempio di forward chain di una tabella filter:

```
-iptables -P FORWARD DROP
-iptables -A FORWARD -i eth0 -m state --state NEW -j ACCEPT
-iptables -A FORWARD -i ppp0 -s 87.15.12.0/24 -m state --state NEW -j ACCEPT
-iptables -I FORWARD 1 -m state --state ESTABLISHED -j ACCEPT
```

La prima regola stabilisce una policy di default impostata come DROP. Nelle successive due regole vengono accettati tutti i pacchetti entranti nell'interfaccia `eth0` e tutti i pacchetti entranti nell'interfaccia `ppp0` con indirizzo sorgente appartenente alla rete 87.15.12.0/24. Per quanto riguarda il controllo dello stato si può notare come le regole accettino in prima istanza solo i pacchetti che inizializzano una connessione (`--state NEW`) e successivamente il controllo effettuerà una policy di ACCEPT a tutti i pacchetti delle connessioni già attive (`--state ESTABLISHED`).

Per effettuare i vari test e poterli confrontare è stato creato uno *script* che genera in automatico N regole di un *firewall stateless*. In particolar modo le regole di filtraggio andranno a controllare gli indirizzi IP sorgente e destinazione, il protocollo e il numero di porta. Quest'ultimo sarà assegnato in maniera casuale. Affinchè il controllo vada a buon fine, è stato necesssario introdurre una "regola di ritorno" alla fine della tabella e una regola di default impostata come DROP.

4.3 Firewall con XDP

Di seguito viene mostrato il codice presente all'interno del file `xdp.c` che implementa un firewall stateless realizzato con XDP ed eBPF.

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

#define SEC(NAME) __attribute__((section(NAME),used))

#define IP2 3232235777  //192.168.1.1 => c0.a8.01.01//
#define IP0 3232235521  //192.168.0.1 => c0.a8.00.01//

SEC("prog")
```

Nella prima parte del programma vengono definite le librerie utilizzzate in seguito per l'analisi dell'intestazione dei pacchetti in transito nel firewall. L'istruzione `SEC("prog")` necessita di una definizione per poter alleggerire la scrittura successivamente e definisce una sezione ELF del codice (già citata nel paragrafo 3.4 inerente ad Iproute2). ELF è un formato file standard per eseguibili, codici oggetto e librerie condivise. In particolar modo, è stato scelto come formato standard dei file binari per i sistemi Unix.

```
int firewall(struct xdp_md *ctx)
{
      struct vlan_ethhdr
      {
            unsigned char   h_dest[ETH_ALEN];
            unsigned char   h_source[ETH_ALEN];
            __be16          h_vlan_proto;
            __be16          h_vlan_TCI;
            __be16          h_vlan_encapsulated_proto;
      };

      void *data_end = (void *)(long)ctx->data_end;
      void *data = (void *)(long)ctx->data;

      struct vlan_ethhdr *vlanhdr = data;

       if (data + sizeof(*vlanhdr) > data_end)
       {
            return XDP_ABORTED;
       }
```

L'intero programma è contenuto all'interno di una funzione nominata `firewall` che accetta come parametro un puntatore ad una struttura che contiene il pacchetto in arrivo dalla porta di rete. Prima di effettuare l'analisi dell'intestazione dei pacchetti, l'eBPF richiede che vengano generati due puntatori `*data` e `*data_end` che indicano rispettivamente l'inizio e la fine del pacchetto. Entrambi i valori sono contenuti all'interno di due campi della struttura `xdp_md`. Il controllo del pacchetto viene effettuato applicando dei puntatori a strutture che contengono "la maschera" dei campi delle diverse intestazioni. Così facendo sarà sufficiente andare a prelevare il contenuto del campo interessato per l'analisi. Il verificatore eBPF richiede che venga effettuato un controllo sulla lunghezza dell'intestazione in modo tale da garantire che il contenuto del puntatore applicato non ecceda rispetto alla lunghezza del pacchetto. Nel caso in cui il controllo risulti errato, il codice restituisce un'errore e il pacchetto viene rilasciato dall'azione XDP_ABORTED (vedi paragrafo 2.10 inerente all'XDP).

```
if(__constant_ntohs(vlanhdr->h_vlan_encapsulated_proto) == ETH_P_ARP)
{
      return XDP_PASS;
}

if(__constant_ntohs(vlanhdr->h_vlan_encapsulated_proto) == ETH_P_IP)
{
      struct iphdr *ip = data + sizeof(*vlanhdr);

      struct tcphdr *tcp = data + sizeof(*vlanhdr) + sizeof(*ip);

      if(data + sizeof(*vlanhdr) + sizeof(*ip) + sizeof(*tcp) > data_end)
```

```
	{
		return XDP_ABORTED;
	}

	struct udphdr *udp = data + sizeof(*vlanhdr) + sizeof(*ip);

	if (data + sizeof(*vlanhdr) + sizeof(*ip) + sizeof(*udp) > data_end)
	{
		return XDP_ABORTED;
	}
```

Per semplicità si è deciso di implementare un firewall stateless, questo significa che il controllo sarà effettuato sui valori chiave che identificano univocamente una connessione:

- Indirizzo IP sorgente e destinazione
- Protocollo (TCP, UDP, ICMP, …)
- Porta sorgente e porta destinazione

Per effettuare i controlli sopra elencati è, quindi, stato necessario andare a generare i puntatori `*vlanhdr`, `*ip`, `*tcp` e `*udp` che contengono le rispettive intestazioni. A seguito di ogni definizione dei puntatori è necessario effettuare nuovamente il controllo sulla fine del pacchetto. Nella sezione di codice precedente è stata implementata una prima verifica sull'ethertype per poter accettare tutti i pacchetti *ARP request* e *ARP reply*. Se invece il pacchetto è di tipo *IP*, viene svolto un'ulteriore controllo sul protocollo (*TCP o UDP*).

```
__be32 src_ip = __constant_ntohl(ip->saddr);
__be32 dst_ip = __constant_ntohl(ip->daddr);
__u8 protocol = ip->protocol;
__u16 src_port;
__u16 dst_port;

if(ip->protocol == IPPROTO_UDP)
{
	src_port = __constant_ntohs(udp->source);
	dst_port = __constant_ntohs(udp->dest);
}
else if (ip->protocol == IPPROTO_TCP)
{
	src_port = __constant_ntohs(tcp->source);
	dst_port = __constant_ntohs(tcp->dest);
}
```

Nell'ultima parte del programma vengono prelevati i diversi campi delle intestazioni e salvati nelle rispettive variabili, che verranno usate nelle regole di filtraggio.

4.3.1 Generazione delle regole

Il firewall implementa una politica di default impostata con XDP_DROP, il programma rifiuta tutti i pacchetti eccetto quelli che rientrano nelle regole specificate. Tali regole vengono realizzate utilizzando una condizione *if*. Se i campi prelevati dall'intestazione del pacchetto agganciato dall' XDP soddisfano tutti le uguaglianze, allora viene effettuata un'azione di XDP_PASS e il kernel passerà il pacchetto all'elaborazione di rete.

```
if (protocol == IPPROTO_TCP && src_ip == (int)IP2 && dst_ip == (int)IP0 &&
src_port == 4444)
{
      return XDP_PASS;
}
```

Per agevolare la scrittura delle regole è stato utilizzato uno *script* che genera un numero di regole stabilito dall’utente. Per differenziare le regole è stata impostata la porta destinazione in maniera casuale con un valore compreso tra 1 e 65535. Prima di terminare il programma è necessasrio aggiungere una “regola di ritorno” per l’ultima generata, andando ad invertire gli indirizzi IP. Così facendo è stato possibile realizzare i test con un numero incrementale di regole di cui esclusivamente l’ultima sarà quella che passerà il controllo.

Capitolo 5

Ambiente di sviluppo dei programmi

In questo capitolo verrà per prima cosa effettuata una panoramica sullo scenario hardware e software a disposizione per effettuare i test. Per avere a disposizione una rete ad alte prestazioni ed elevati throughput, sono stati utilizzati alcuni server di CloudLab. Successivamente verranno presentati i risultati dei test effettuati, con particolare attenzione alle differenze tra il *packet processing* effettuato nel Kernel e quello sulle macchine virtuali presenti nello spazio utente.

5.1 CloudLab

CloudLab è un'ambiente di sviluppo finanziato dalle università americane che consente ai ricercatori di sperimentare architetture cloud. È progettato per l'esecuzione di esperimenti che porteranno a nuove funzionalità nei cloud futuri o ad una comprensione più approfondita dei fondamenti del cloud computing. Ciò significa che è ideale per esperimenti che non possono essere eseguiti su cloud tradizionali perché richiedono controllo e/o visibilità su parti del sistema come i livelli di virtualizzazione, archiviazione o rete. Attualmente sono attivi tre progetti in cui l'architettura di base di ogni sito presenta circa 5000 core e 500 terabyte di spazio di archiviazione compatibile con la virtualizzazione. CloudLab può allocare diversi ambienti di ricerca, permette quindi ai ricercatori di configurare il proprio ambiente di lavoro secondo le necessità richieste.

Grazie alla collaborazione tra i gruppi di ricerca delle Università, per effettuare i diversi test è stata utilizzata una rete appartenente al progetto CloudLab Utah formata da 3 server connessi da uno switch.

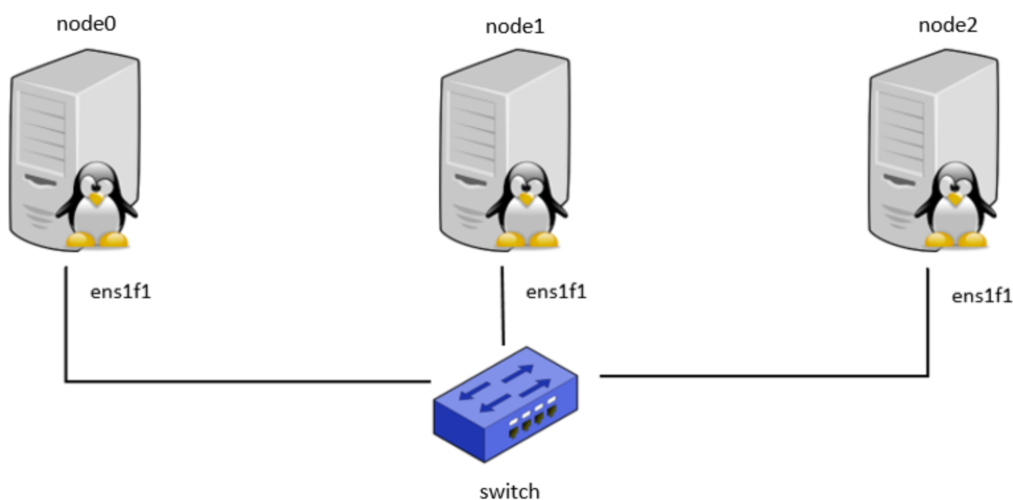

Figura 8: Connessione fisica dei nodi della rete

I server a disposizione per efettuare i test sono HPE ProLiant XL170r, con 10 core ciascuno, e presentano le caratteristiche presenti nella tabella sottostante.

XL 170	Intel Broadwell, 10 cores, 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)

Tabella 1: Caratteristiche dei server CludLab utilizzati

Ogni server è connesso, tramite un collegamento sperimentale a 25 Gbps, agli switch Mellanox 2410.

Su questa rete sono state generate due VLAN in modo da modificare la configurazione della rete e rimuovere il collegamento diretto tra *node0* e *node2*. Grazie a tale connessione è possibile implementare il firewall sul *node1* della rete e quindi andare a testare la funzione di rete. Così facendo tutto il traffico dovrà passare attraverso il firewall.

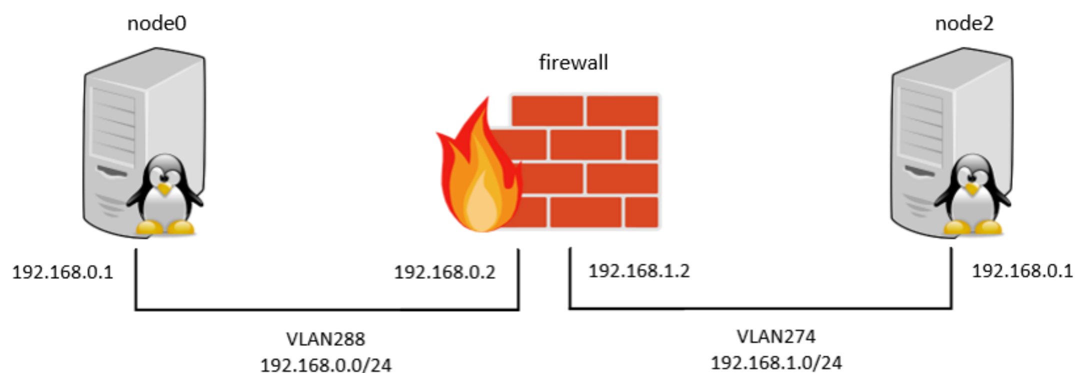

Figura 9: Collegamenti realizzati con le VLAN sullo switch

I dati raccolti dai test effettuati sono stati realizzati in tre diversi scenari principalmente:

- Firewall XDP: realizzato tramite un programma su macchina virtuale eBPF e caricato sulla scheda di rete come descritto nel Capitolo 3
- Firewall Iptables: realizzato tramite uno *script* che genera automaticamente N regole in maniera automatica e con numero di porta destinazione casuale esattamente come per il programma XDP
- Firewall Iptables su VM: realizzato come il caso precedente, ma all'interno di una macchina virtuale installata sul nodo della rete
    - Macchina Virtuale Virtio
    - Macchina Virtuale e1000

5.2 Macchine Virtuali per lo sviluppo del kernel bypass

Per testare le funzionalità del firewall all'interno dello spazio utente sono state utilizzate due diverse *virtual machine*. La virtualizzazione permette di astrarre gli elementi hardware (hard disk, RAM, CPU, interfacce di rete, …) e renderli disponibili sotto forma di risorse virtuali.

Per testare le prestazioni del kernel bypass è stata utilizzata una virtualizzazione delle interfacce di rete nominata E1000. Essa fornisce l'emulazione dell'adattatore di rete Intel 82545EM Gigabit Ethernet (E1000). In particolar modo E1000 effettua la cosiddetta "virtualizzazione completa" dell'hardware ovvero l'Hypervisor provvederà ad emulare un sistema hardware completo e standardizzato rendendo trasparente alla VM il fatto di trovarsi all'interno di una infrastruttura virtualizzata o meno. L'Hypervisor si può considerare come quello strato aggiuntivo che si interpone tra l'hardware della macchina fisica e le macchine virtuali e che ne controlla e gestisce il traffico delle informazioni.

Si è voluto testare anche la differenza tra la virtualizzazione e la paravirtualizzazione, andando ad installare sulla rete una macchina virtuale chiamata Virtio. La paravirtualizzazione non crea un'emulazione dell'hardware di un generico computer, ma sarà compito dell'Hypervisor andare a controllare e regolamentare l'accesso all'hardware sottostante da parte delle VM. Questa tecnica permette di raggiungere prestazioni pari a quelle di un ambiente non virtualizzato. Un requisito necessario per l'utilizzo di questa macchina virtuale è che il sistema operativo implementato sia compatibile con l'ambiente fisico, a differenza dell'E1000 per il quale non è requisito necessario al corretto funzionamento.

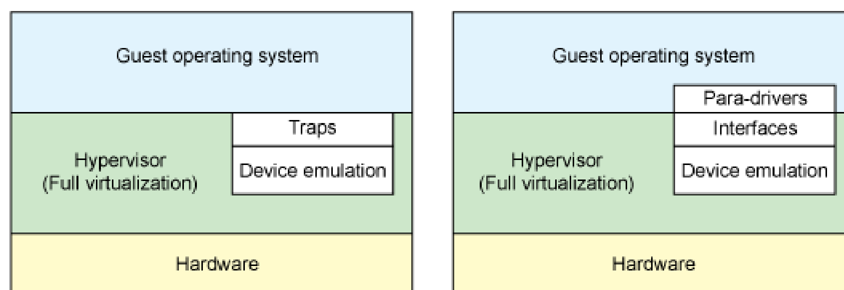

Figura 10: Architetture VM con virtualizzazione completa (a sinistra) e paravirtualizzazione (a destra)

Quando siamo in un caso di virtualizzazione completa l'hypervisor deve emulare i comportamenti dell'hardware reale. Sebbene ciò offra la massima flessibilità (eseguendo un sistema operativo non modificato), introduce inefficienza. Il lato destro della figura 10 mostra il caso di paravirtualizzazione in cui il sistema operativo *guest* è consapevole di essere in esecuzione su un hypervisor e include i driver che fungono da *front-end*. L'hypervisor implementa i driver *back-end* per la particolare emulazione del dispositivo. Questi driver sono all'ingresso della VM Virtio, fornendo un'interfaccia standardizzata per lo sviluppo dell'accesso al dispositivo emulato per propagare il riutilizzo del codice e aumentare l'efficienza.

Per la realizzazione dei firewall sulle VM è stato utilizzato lo stesso *script* di Iptables per generare N regole con numero di porta casuale.

5.3 Test sulla larghezza di banda

Iperf3 è un tool Linux in grado di realizzare misure attive della massima larghezza di banda ottenibile sulla rete IP. È possibile impostare diversi parametri relativi al tempo e al tipo di protocollo dei pacchetti che verranno generati. Per ogni test esso riporta oltre alla larghezza di banda anche la quantità di traffico generato ed il numero di pacchetti ritrasmessi. Nella porzione di codice è mostrato un esempio di risultato realizzato.

```
[ ID] Interval           Transfer     Bandwidth       Retr
[  4]   0,00-100,00 sec   187 GBytes  16,0 Gbits/sec  4438             sender
[  4]   0,00-100,00 sec   187 GBytes  16,0 Gbits/sec                  receiver
```

Per misurare il massimo della banda e quindi la massima velocità di processamento dei pacchetti sulla linea, il traffico generato è notevolmente maggiore della capacità del canale.

Tutti i risultati dei divesi test sono prodotti da un traffico TCP per una durata di 100 secondi impostando il *node2* della rete come server, attraverso il comando:

```
iperf3 -s -p <porta>
```

Affinchè il test produca un risultato è stato necessario impostare manualmente il numero di porta dell’ultima istruzione accettata da firewall la quale sarà anche l’unica porta che accetta i pacchetti di ritorno.

Il *node0* impostato come *client* andrà a generare il traffico secondo le specifiche elencate.

```
iperf3 -c 192.168.1.1 -p <porta> -t 100
```

Tutte le misure relative ai risultati dei test mostrati nei paragrafi sono state realizzate mediante Iperf3.

5.3.1 Confronto tra XDP, Iptables e VM

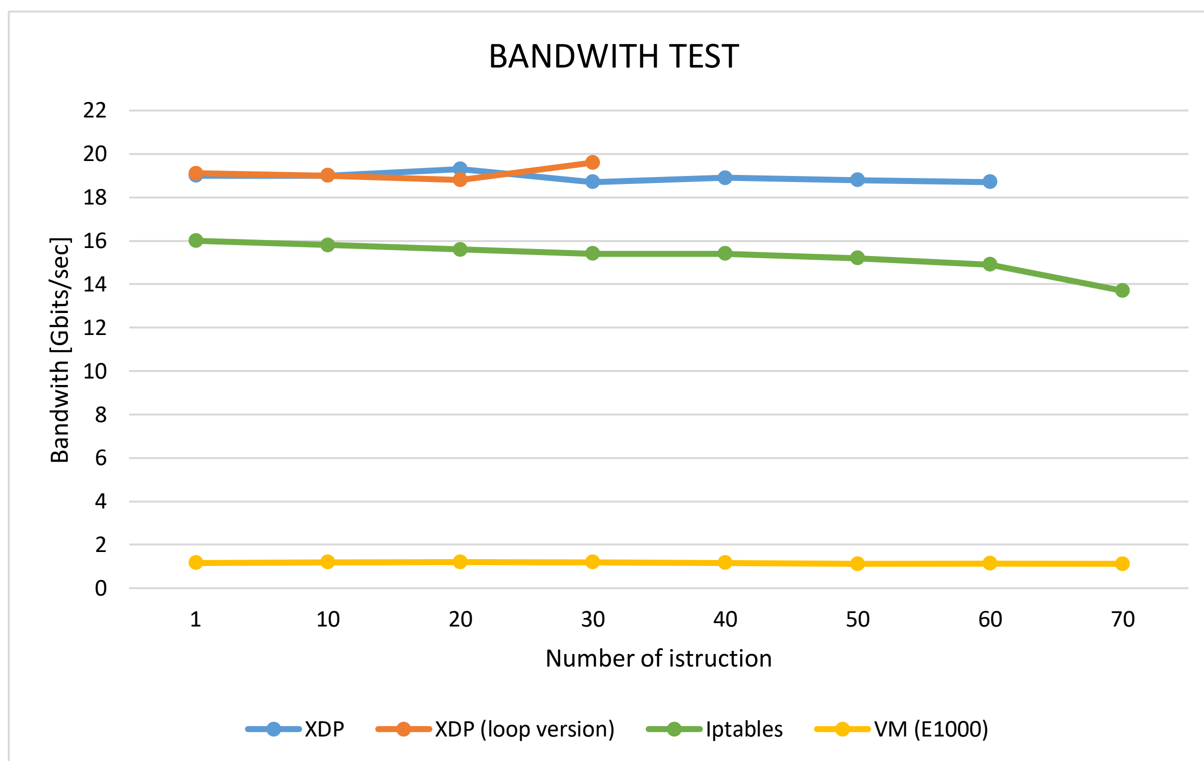

Figura 11: Grafico che mostra l'andamento della Bandwith all'aumentare del numero di regole

La tabella mostra i valori ottenuti indicando, per ogni firewall testato, il valore della larghezza di banda in Gbits/sec ed il numero totale dei pacchetti ritrasmessi in funzione del numero di regole generate. In realtà le regole generate sono quelle indicate in tabella con l'aggiunta della "regola di ritorno" e la regola di default.

	XDP		XDP (loop version)		Iptables		VM (E1000)	
	Bandwith	Retr	Bandwith	Retr	Bandwith	Retr	Bandwith	Retr
1	19	5466	19,1	3644	16	4438	1,17	89
10	19	4176	19	5636	15,8	4422	1,19	91
20	19,3	12129	18,8	2975	15,6	4820	1,2	131
30	18.7	3182	19,6	10675	15,4	4834	1,19	99
40	18.9	2116	-	-	15,4	4399	1,17	92
50	18,8	2975	-	-	15,2	4633	1,12	86
60	18.7	869	-	-	14,9	4801	1,13	132
70	-	-	-	-	13,7	2705	1,12	90

Tabella 2: Risultati ottenuti dal test sulla Bandwith

Per quanto riguarda il test sulla larghezza di banda è stata implementata una seconda versione del programma XDP che genera le regole attraverso un ciclo *loop* andando ad impostare ad ogni iterazione il numero di porta pari all'indice del ciclo. Dal grafico si può notare come questa soluzione funzioni correttamente entro un numero di cicli, limitato dal *BPF verifier* (descritto nel paragrafo 2.9). Provando a generare più regole il programma restituisce un errore dato dal verificatore, il quale effettuando il *loop unrolling* non garantisce una fine al programma.

Implementando il programma XDP senza l'utilizzo del loop è possibile notare che si riesce a superare le 30 regole di filtraggio. Ma già con 70 regole si eccede con le dimensioni stesse del programma in memoria, andando a superare i 512 Bytes previsti per i programmi eBPF.

```
error: Looks like the BPF stack limit of 512 bytes is exceeded. Please move
large on stack variables into BPF per-cpu array map.
```

Come suggerisce l'errore stesso è possibile aggirare questa problematica utilizzando le mappe che, essendo condivise con lo spazio utente, hanno uno spazio limite superiore.

5.3.2 Differenza tra E100 e Virtio

Un altro test effettuato vuole porre il confronto tra la virtualizzazione e la paravirtualizzazione rispetto all'hardware nativo. Il confronto è stato effettuato andando ad implementare lo stesso firewall con Iptables.

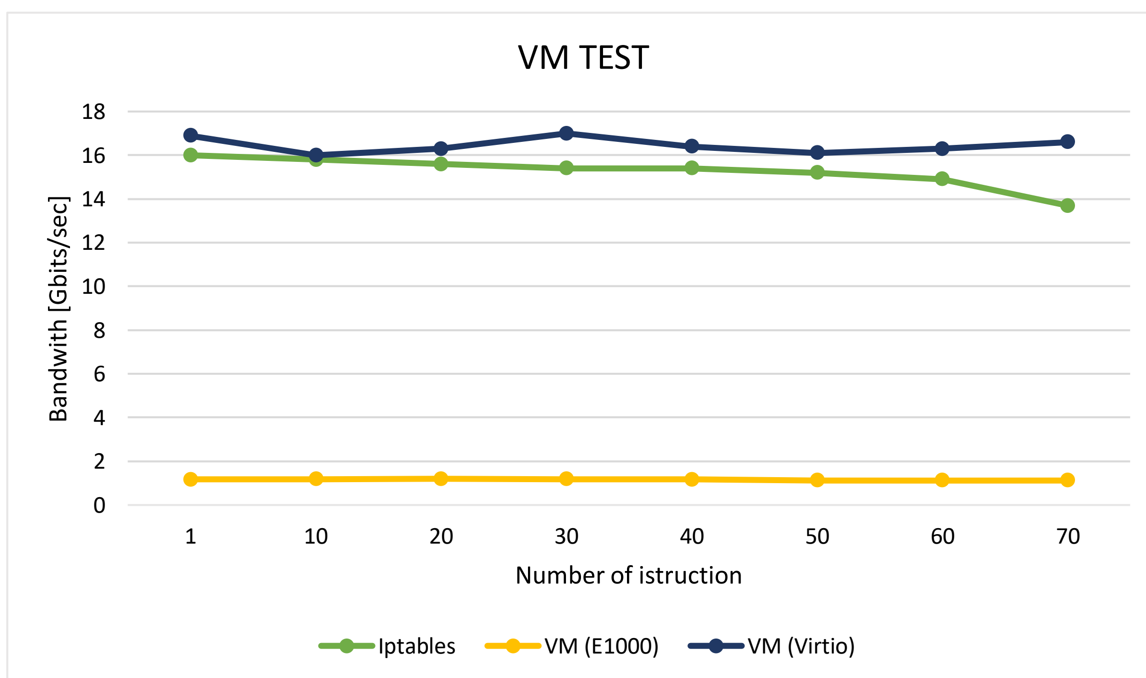

Figura 12: Confronto tra le prestazioni delle VM ed Iptables

	Iptables		VM (E1000)		VM (Virtio)	
	Bandwith	Retr	Bandwith	Retr	Bandwith	Retr
1	16	4438	1,17	89	16,9	11661
10	15,8	4422	1,19	91	16	16150
20	15,6	4820	1,2	131	16,3	13548
30	15,4	4834	1,19	99	17	8105
40	15,4	4399	1,17	92	16,4	10273
50	15,2	4633	1,12	86	16,1	16024
60	14,9	4801	1,13	132	16,3	17936
70	13,7	2705	1,12	90	16,6	13892

Tabella 3: Risultati ottenuti dal test della Banwith delle VM

Come si può notare dai risultati ottenuti andando a confrontare le due macchine virtuali i valori della larghezza di banda sono notevolemente differenti. Tenendo conto della variabilità dei dati e non del singolo test effettuato e considerando i valori del firewall Iptables come le prestazioni dell'hardware nativo, si può affermare che la paravirtualizzazione (VM Virtio) funziona alla pari dell'hardware che la supporta. La possibilità che sia addirittura migliore può risiedere nel fatto che Iptables non è la migliore implementazione della funzione di rete, come è stato confermato dal test precedente.

5.3.3 Numero di regole elevato

Come ultimo test si è voluto verificare l'andamento delle prestazioni delle due migliori alternative all'XDP, effettuando un'incremento notevole di regole.

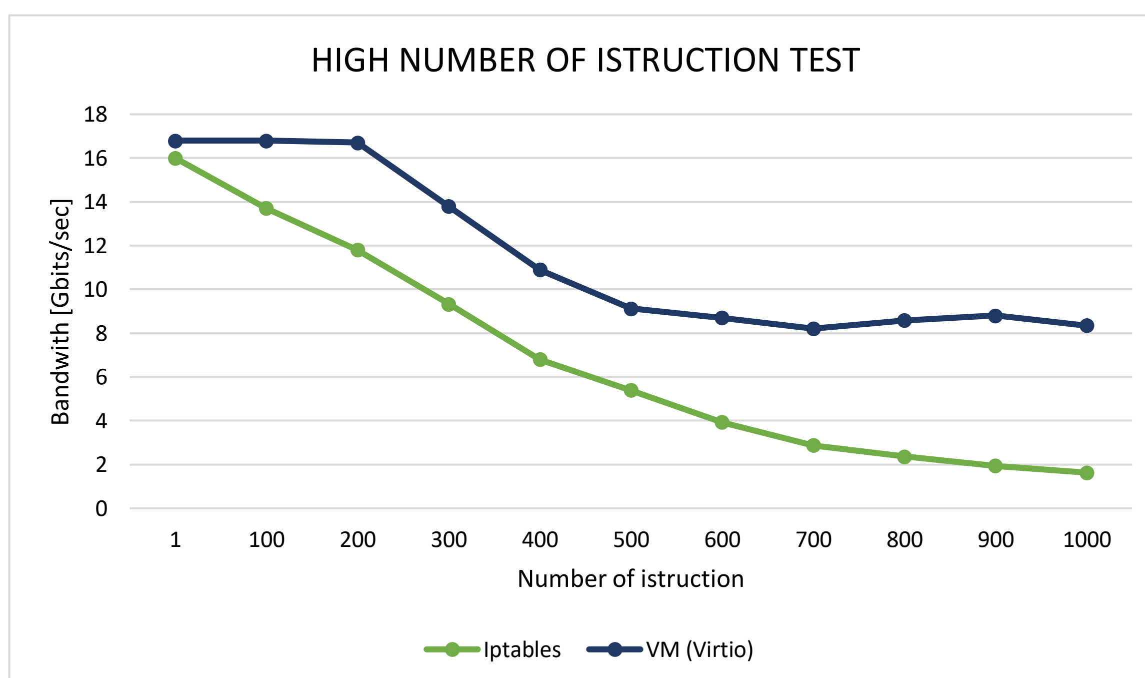

Figura 13: Test sulla larghezza di banda con un numerodi istruzioni elevato

Come risulta dal grafico di Figura 13 andando ad incrementare notevolmente il numero delle istruzioni con Iptables si arriva ad avere delle larghezze di banda molto basse. Anche utilizzando la VM Virtio, che sfrutta la paravirtualizzazione, incrementando il numero di regole si nota un calo delle prestazioni, anche se non così elevato.

Capitolo 6

Conclusioni

In questo elaborato è stato presentato un esempio di programma XDP per eBPF con lo scopo di andare a verificare l'elaborazione veloce di pacchetti programmabili nel kernel.

Le valutazioni fatte a seguito dei test hanno evidenziato un notevole aumento delle prestazioni di XDP che riesce a raggiungere una larghezza di banda di circa 20 Gbits/sec, rispetto ad Iptables che si aggira attorno ai 16 Gbits/sec.

Si può inoltre affermare che l'elaborazione all'interno del kernel è sensibilmente migliore rispetto alle tecniche di *kernel bypass*, ad eccezione delle macchine virtuali che implementano la paravirtualizzazione. Andando però a testare la funzione di rete con un'elevato numero di regole, si può notare dai risultati di figura 13 che le prestazioni calano notevolemente.

Dagli esiti del test effettuato si può affermare che XDP possiede le prestazioni più elevate in termini di rapidità di processamento e quindi di larghezza di banda, grazie alla modalità di "aggancio del pacchetto" da parte degli *hook*. Il verificatore eBPF garantisce la sicurezza del kernel andando a porre particolari vincoli nei programmi, limitanto notevolmente le potenzialità delle funzioni di rete. Queste caratteristiche hanno creato la necessità di una soluzione alternativa, la quale risiede nelle mappe eBPF e nelle funzioni *helper*. Si ritiene quindi possibile la realizzazione delle diverse funzioni di rete andando a sfruttare le potenzialità delle mappe eBPF. All'interno dell'appendice B viene mostrato un'accenno verso questa nuova possibile implementazione del programma.

Appendice A

In questa appendice viene presentato per intero il codice del programma XDP descritto nel paragrafo 4.3 con una sola regola di filtraggio sulla porta 4444.

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

#define SEC(NAME) __attribute__((section(NAME),used))

static int (*bpf_trace_printk)(const char *fmt, int fmt_size, ...) = (void *)
BPF_FUNC_trace_printk;
#define printt(fmt, ...)                                                    \
            ({                                                              \
                char ____fmt[] = fmt;                                       \
                bpf_trace_printk(____fmt, sizeof(____fmt), ##__VA_ARGS__); \
            })


#define IP2 3232235777  //192.168.1.1 => c0.a8.01.01//
#define IP0 3232235521  //192.168.0.1 => c0.a8.00.01//

SEC("prog")

int firewall(struct xdp_md *ctx)
{

struct vlan_ethhdr
{
unsigned char   h_dest[ETH_ALEN];
             unsigned char   h_source[ETH_ALEN];
             __be16          h_vlan_proto;
             __be16          h_vlan_TCI;
             __be16          h_vlan_encapsulated_proto;
};

void *data_end = (void *)(long)ctx->data_end;
void *data = (void *)(long)ctx->data;

       struct vlan_ethhdr *vlanhdr = data;

       if (data + sizeof(*vlanhdr) > data_end)
       {
            return XDP_ABORTED;
       }

       if(__constant_ntohs(vlanhdr->h_vlan_encapsulated_proto) == ETH_P_ARP)
       {
            return XDP_PASS;
       }

       if(__constant_ntohs(vlanhdr->h_vlan_encapsulated_proto) == ETH_P_IP)
       {
```

```
            struct iphdr *ip = data + sizeof(*vlanhdr);

            struct tcphdr *tcp = data + sizeof(*vlanhdr) + sizeof(*ip);

            if (data + sizeof(*vlanhdr) + sizeof(*ip) + sizeof(*tcp) >
data_end)
            {
                      return XDP_ABORTED;
            }

            struct udphdr *udp = data + sizeof(*vlanhdr) + sizeof(*ip);

if (data + sizeof(*vlanhdr) + sizeof(*ip) + sizeof(*udp) > data_end)
            {
                      return XDP_ABORTED;
            }

            __be32 src_ip = __constant_ntohl(ip->saddr);
            __be32 dst_ip = __constant_ntohl(ip->daddr);
            __u8 protocol = ip->protocol;
            __u16 src_port;
            __u16 dst_port;

            if(ip->protocol == IPPROTO_UDP)
            {
                      src_port = __constant_ntohs(udp->source);
dst_port = __constant_ntohs(udp->dest);
}
            else if (ip->protocol == IPPROTO_TCP)
            {
                   src_port = __constant_ntohs(tcp->source);
                   dst_port = __constant_ntohs(tcp->dest);
            }

            if (protocol == IPPROTO_TCP && src_ip == (int)IP2 && dst_ip ==
(int)IP0 && src_port == 4444)
return XDP_PASS;
            if (protocol == IPPROTO_TCP && src_ip == (int)IP0 && dst_ip ==
(int)IP2 && dst_port == 4444)
                   return XDP_PASS;

            return XDP_DROP;
}
       else
{
           return XDP_ABORTED;
}
}

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

Appendice B

Per questo programma XDP per eBPF **non** sono state utilizzate le mappe. In fase di test è risultato inoltre che questa possibile implementazione del programma non è stata la più efficiente, andando a confermare le limitazioni date dal verificatore. La macchina virtuale eBPF, a differenza del BSD, trae molto vantaggio nell'implementazione delle mappe e del loro uso nei diversi programmi. Si può quindi affermare che sarebbe possibile modificare il programma ponendo le regole di filtraggio direttamente nelle mappe.

Grazie ad un *tool,* nominato appunto Bpftool, installato sulla macchina Linux è possibile generare, modificare ed eliminare le mappe.

In particolar modo è possibile generare le mappe esclusivamente dallo spazio utente. Solo in seguito, all'interno del programma eBPF, sarà possibile andare ad inizializzarle con la seguente scrittura.

```
struct bpf_map_def {
    unsigned int type;
    unsigned int key_size;
    unsigned int value_size;
    unsigned int max_entries;
};

struct bpf_map_def SEC("maps") firewall_map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(__u32),
    .max_entries = 60000,
};
```

Nell'esempio la mappa è di tipo ARRAY con chiavi e valori di 4 Byte. Si può notare inoltre che la mappa possiede 60000 elementi e quindi nello spazio utente verrà allocata la memoria necessaria a contenere una mappa con queste caratteristiche.

Pur risiedendo nello spazio utente, le mappe condivise con il kernel non andranno ad influire nelle prestazioni.

Nel kernel sono definiti alcuni *helper* dedicati per la gestione delle mappe. Selezionando una chiave di una determinata mappa sarà possibile andare a prelevare il valore, aggiornare il valore o eliminare la coppia chiave/valore.

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

int bpf_map_update_elem(struct bpf_map *map, const void *key, const void *value,
u64 flags)

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

Per poter usufruire delle funzioni descritte è necessario andare a dichiararle all'interno del programma eBPF.

Bibliografia e Sitografia

1. *“The BSD Packet Filter: A New Architecture for User-level Packet Capture”*, Steven McCanne and Van Jacobson, Lawrence Berkeley Laboratory, December 19, 1992

2. A thorough introduction to eBPF, Matt Fleming, https://lwn.net/Articles/740157/, December 2, 2017

3. BPF, eBPF, XDP and Bpfilter… What are These Things and What do They Mean for the Enterprise?, Nic Viljoen, https://www.netronome.com/blog/bpf-ebpf-xdp-and-bpfilter-what-are-these-things-and-what-do-they-mean-enterprise/, Apr 16, 2018

4. BPF and XDP Reference Guide, https://cilium.readthedocs.io/en/latest/bpf/

5. https://github.com/iovisor/bpf-docs/blob/master/bpf_helpers.rst

6. https://www.iovisor.org/technology/xdp

7. Express Data Path, Brenden Blanco and Tom Herbert, https://en.wikipedia.org/wiki/Express_Data_Path, 2016

8. Høiland-Jørgensen, pubblicato il 2019-05-03, Source text and experimental data for our paper describing XDP: tohojo/xdp-paper

9. The LLVM Compiler Infrastructure, https://llvm.org/, 1 August 2019

10. Firewall, https://it.wikipedia.org/wiki/Firewall

11. Iproute2, Alexey Kuznetsov and Stephen Hemminger, https://en.wikipedia.org/wiki/Iproute2, July 8, 2019

12. *“Prototipazione di servizi di rete distribuiti con eBPF”*, Francesco Picciariello, Aprile 2018

13. *“Elaborazione dei pacchetti nel kernel Linux tramite eXpress Data Path (XDP)”*, Fabrizio Finelli, Anno accademico 2018/2019

14. The CloudLab Manual Hardware, https://docs.cloudlab.us/hardware.html

15. What Does it Mean to Build a Cloud on Cloud Lab? , https://cloudlab.us/

16. Network functions virtualization, https://it.wikipedia.org/wiki/Network_functions_virtualization

17. Compilatore just-in-time, https://it.wikipedia.org/wiki/Compilatore_just-in-time

18. vSphere: elementi caratterizzanti di una macchina virtuale, Alessio Carta, https://www.guruadvisor.net/it/vmware/608-vsphere-elementi-caratterizzanti-di-una-macchina-virtuale

19. Virtualizzazione e paravirtualizzazione, pro e contro rispetto alle soluzioni tradizionali, Riccardo Riva, https://www.hostingtalk.it/virtualizzazione-e-paravirtualizzazione-pro-e-contro-rispetto-alle-soluzioni-tradizionali_-c000000wo/

20. Virtio: An I/O virtualization framework for Linux, M. Jones, https://developer.ibm.com/articles/l-virtio/, January 29, 2010