BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

PACKET FILTERING USING XDP

FILTROVÁNÍ PAKETŮ POMOCÍ XDP

TERM PROJECT
SEMESTRÁLNÍ PROJEKT

AUTHOR **Bc. JAKUB MACKOVIČ**
AUTOR PRÁCE

SUPERVISOR **Ing. MATĚJ GRÉGR, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Information Systems (DIFS) Academic year 2018/2019

Master's Thesis Specification

21433

Student:	**Mackovič Jakub, Bc.**
Programme:	Information Technology Field of study: Information Systems
Title:	**Packet Filtering Using XDP**
Category:	Networking

Assignment:

1. Get familiar with eXpress Data Path and eBPF technologies.
2. Design a packet filtration mechanism using XDP and eBPF.
3. Implement the design and deploy it in Brno University of Technology networking testbed.
4. Test the performance of your approach and compare it with iptables and ipset tools.

Recommended literature:

- Bharadwaj, R. (2017). Mastering Linux Kernel development: A kernel developer's reference manual. ISBN: 978-1-78588-613-3.
- Robert Love. 2010. Linux Kernel Development (3rd ed.). Addison-Wesley Professional. ISBN: 978-0-672-32946-3

Requirements for the semestral defence:

- Items 1 and 2

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor:	**Grégr Matěj, Ing., Ph.D.**
Head of Department:	Kolář Dušan, doc. Dr. Ing.
Beginning of work:	November 1, 2018
Submission deadline:	May 22, 2019
Approval date:	October 31, 2018

Packet Filtering Using XDP

Declaration

I hereby declare that this thesis was prepared as an original author's work under the supervision of Mr. Ing. Matěj Grégr, PhD. All the relevant information sources which were used during the preparation of this thesis are properly cited and included in the list of references.

. . . .
Jakub Mackovič
May 21, 2019

Acknowledgements

I would like to thank Mr. Grégr for his professional supervision, for the continuous support that he has provided me for the duration of writing this thesis and for his constructive ideas.

Abstract

Computer systems which must provide their services with a high availability require certain security measures to remain available even when under packet-based network attacks. Unwanted packets must be dropped or mitigated as early as possible and as quickly as possible. This work analyses the eXpress Data Path (XDP) as a technique for early packet dropping and the extended Berkeley Packet Filter (eBPF) as a mechanism for high-speed packet analysis. Examples of current firewalling practices on Linux kernel based systems are observed and a design and the behavioural goals of a system for high-speed packet filtering based on eBPF and XDP are provided. The implementation of the design is then described in detail. Finally, results of several performance tests are presented, showing the XDP solution's performance advatages over contemporary filtering techniques.

Abstrakt

Počítačové systémy, ktoré musia poskytovať svoje služby s vysokou dostupnosťou vyžadujú isté bezpečnostné opatrenia na to, aby ostali dostupné aj pod paketovými sieťovými útokmi. Nevyžiadané pakety musia byť zahodené čo najskôr a čo najrýchlejšie. Táto práca analyzuje eXpress Data Path (XDP) ako techniku skorého zahodenia paketov a extended Berkeley Packet Filter (eBPF) ako mechanizmus rýchlej analýzy obsahu packetov. Poskytuje sa pohľad na dnešnú prax v oblasti firewallov v systémoch s linuxovým jadrom a navrhne sa systém rýchlej filtrácie paketov založený na eBPF a XDP. Do detailov popisujeme naimplementované filtračné riešenie. Nakoniec sa vyzdvihujú výhody XDP oproti ostatným súčasným technikám filtrácie paketov na sérii výkonnostných testov.

Keywords

XDP, BPF, eBPF, packet filtering, NETX

Klíčová slova

XDP, BPF, eBPF, filtrovanie paketov, NETX

Reference

MACKOVIČ, Jakub. *Packet Filtering Using XDP*. Brno, 2019. Term project. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Matěj Grégr, Ph.D.

Rozšířený abstrakt

Od počiatku medzipočítačovej komunikácie a od vynájdenia celosvetovej siete Internet sme svedkami čoraz väčšej záťaže sieťových prvkov nielen kvôli prudko rastúcej popularite streamovaných služieb, ale aj kvôli rastúcim počtom útokov na počítačové siete. Je preto dôležité, aby počítače, ktoré na Internete poskytujú kritické služby, vydržali záťaž ako požiadavkov na službu, tak sieťových útokov bez toho, aby prestali svoje služby ponúkať.

V tejto práci sa zaoberáme vývinom systému, ktorý by napomohol počítačom bežiacim na dostatočne novom linuxovom jadre so znižovaním účinku sieťových útokov, primárne distribuovaného typu. Ako zvolená je pomerne nová technológia XDP, ktorá definuje isté rozhranie pre ovládače sieťových kariet pre spracovávanie paketov ešte pre tým, než sa z pamäti sieťovej karty nakopírujú do jadra. To rozhranie umožňuje zavolať filtračný program interpretovaný virtuálnym strojom extended Berkeley Packet Filteru.

Najprv skúmame súčasné riešenia firewallov v počítačoch založených na linuxovom jadre. Kladieme dôraz na nástroj `iptables`, ktorý dokáže naplniť takzvané *chains* pomerne zložitými filtračnými pravidlami. Tieto chains sú konzultované v rozdielnych momentoch životného cyklu paketov v operačnom systéme. Cez `PREROUTING` a `POSTROUTING` chainy prechádzajú pakety, ktoré do počítača vchádzajú alebo z neho cez niektoré rozhranie vychádzajú, `INPUT` a `OUTPUT` chainy sú určené pre pakety, ktoré buď pochádzajú z daného počítača alebo sú preň určené, a nakoniec `FORWARD` chain je určený pre pakety, ktoré nie sú určené pre daný počítač a sú ním smerované. V `iptables` sú definované tabuľky *filter*, *nat*, *mangle*, *raw* a *security*. Každá z nich obsahuje svoje instancie niektorých zo spomenutých chainov.

Tento nástroj umožňuje zadávať pravidlá po jednotlivých IP adresách. Preto bol ako doprovodný program k `iptables` vyvinutý nástroj `ipset`, ktorý umožňuje deklarovať rozsahy adries. Tieto rozsahy špecifikované v zadávaných pravidlách nahrádzajú pôvodne veľké množstvá potrebných pravidiel pre vytvorenie zhody pre celé rozsahy adries.

V krátkosti spomíname projekt `netfilter`, ktorý má výkonnosťou prevyšovať `iptables` a zároveň má byť jeho oficiálnym nástupcom. Je však založený na ňom, hlavne použitím chainov.

Následne definujeme Berkeley Packet Filter (BPF) ako základný kameň pre rýchle spracovanie paketov. Modelom filteru BPF je acyklický orientovaný graf. Tento graf obsahuje jeden prvotný uzol, dva cieľové uzly (prijatie a neprijatie paketu) a zvyšné rozhodovacie uzly. Necieľové uzly sú abstrahované jednoduchým predikátom nad niektorým poľom paketu. Ak je tento predikát pravdivý, pokračuje sa uzlom pravým ak je nepravdivý, tak ľavým, až kým tento proces nepríde k cieľovému uzlu. V implementácii predstavuje každý uzol sériu jednoduchých inštrukcií načítania dát na adrese a podmieneného skoku a cieľové uzly predstavujú návratové hodnoty boolovského typu *true* a *false*.

Systém BPF bol pridaný do linuxového jadra vo verzii 2.5. Odvtedy sa na ňom nevykonávali žiadne zmeny ani vylepšenia, až kým vo verzii 3.15 bolo vydané jeho rozšírenie s názvom *extended BPF* (eBPF). Toto rozšírenie reflektuje zmeny v modernom hardware; Šírka registrov BPF bola rozšírená na 64 bitov, ich počet bol zvýšený z dva na desať a bolo umožnené volať istú množinu pomocných funkcií kernelu.

Ďalšie vylepšenia, s ktorými eBPF prišlo boli lepší preklad z virtuálnych inštrukcií eBPF na inštrukcie danej architektúry hardware, statická verifikácia programov zakazujúca napríklad cykly alebo dereferenciu uživateľsky zadaných ukazateľov, systémové volanie BPF, či takzvané mapy, nové dátové typy, ktoré majú eBPF programy k dispozícii. Mapy sú dátové štruktúry typu kľúč-hodnota. Existuje viacero typov týchto máp, ako pole či

hashovacia tabuľka, no z hľadiska implementácie filtračného programu nás najviac zaujal typ strom *Longest Prefix Match*.

Ďalej je popísané XDP ako rozhranie pre ovládače sieťových rozhraní, ktoré môžu pomocou neho zavolať načítaný eBPF (XDP) program. Ten XDP program má k dispozícii dáta celého príchodizeho paketu, na základe ktorých ako návratovú hodnotu vstupnej funkcie vráti verdikt nad daným paketom. Tým verdiktom môže byť zahodenie paketu, posunutie paketu ďalej kernelu, poslanie paketu von na sieť vstupným rozhraním, či presmerovanie paketu na iné rozhranie. XDP môže pracovať v jednom z troch módov, a to buď v natívnom móde, kde XDP program sa nachádza v ovládači, paket sa do jadra nekopíruje a inštrukcie vykonáva procesor počítača, alebo v *offloaded* móde, kde inštrukcie sú vykonávané samotnou sieťovou kartou, alebo v generickom móde, pre ktorý nie je potrebná podpora ovládača a je určená primárne pre účely vývoja.

Z predošlých poznatkov bol navrhnutý filtračný systém, ktorého jadrom je eBPF program vložený do sieťovej karty, volaný ovládačom pomocou XDP rozhrania. Cieľový operačný systém pre túto filtráciu je platforma routerov NETX vyvíjaná na pôde univerzity Vysokého Učení Technického v Brně. Definujeme isté požiadavky na výkonnosť riešenia a na použitie vhodných vlastností, ktoré eBPF a XDP prinášajú.

Následne je do detailu popísaný implementovaný filtračný mechanizmus. Popisujeme, ako sa daný XDP program implementovaný v jazyku C prekladá v dvoch krokoch prekladačmi *clang* a *llc* zo súboru prekladačov projektu Low Level Virtual Machine (LLVM). Druhý program tohoto riešenia je určený na zavádzanie a odstraňovanie XDP programu zo sieťovej karty a na manipuláciu s eBPF mapami. Tento program je takisto dopodrobna popísaný spolu s jeho rozhraním na príkazovom riadku. Potom popisujeme implementáciu samotného XDP programu a jeho logiku spracovania paketov. eBPF mapy obsahujú jednak informáciu o cieľovom výstupnom rozhraní, ako aj pravidlá pre samotnú filtráciu. Do filtračnej mapy sa smie zadať rozsah IP a verdikt, ktorý sa vykoná nad paketmi padajúcimi do toho rozsahu. Ak príchodzí paket nepadá do žiadneho z rozsahov v mape, použije sa implicitné pravidlo presmerovať paket na výstupné rozhranie.

Ďalej sa zameriavame na vykonané výkonnostné testy. XDP program bol porovnávaný s ekvivalentnými nastaveniami v `iptables`, `ipset` a smerovacej tabuľke operačného systému, na ktorom prebiehali testy. Popisujeme metodológiu testov, čiže ako bol umelý tok dát generovaný a ako bol vykonaný zber dát (hlavne miera spracovaných paketov za sekundu). Takisto popisujeme stroje, ktoré hrali isté role pri testovaní (generátor, stroj s filterom a cieľový stroj), ich zapojenie a špecifikáciu hardware. Vrámci zamýšľania sa nad očakávanými výsledkami predpokladáme, že implementovaný XDP program bude mať o jeden rád lepšiu výkonnosť v počte spracovaných paketov za sekundu.

Výsledky sú prezentované v štyroch rôznych scenároch, jeden na zahadzovanie paketov, jeden na ich presmerovanie a dva na meranie času inicializácie filtračných techník, teda času naplnenia ich dátových štruktúr pre filtračné pravidlá. V prvých dvoch prípadoch sa taktiež meria výkonnosť na jedno jadro procesoru.

Výkonnostnými testami na jedno jadro procesoru sa ukázalo, že XDP bolo schopné zahadzovať pakety rýchlosťou až 5,4 miliónov paketov za sekundu (Mpps), respektíve rýchlosťou 2,2 Mpps s prístupom do filtračnej mapy s jedným záznamom. Ostatné riešenia nepresiahli ani milión paketov za sekundu. Jedno jadro XDP bolo schopné presmerovávať pakety rýchlosťou až 2,15 Mpps, respektíve rýchlosťou 0,62 Mpps s jedným prístupom do filtračnej mapy. Smerovací systém linuxového jadra dosiahol sotva 0.32 Mpps. V prípade využitia všetkých 32 jadier procesoru testovacieho stroja dosahuje XDP výkonnosť približne 11 Mpps ako pre zahadzovanie tak pre presmerovanie. V prvom prípade sa mu výkonnostne vyrovanávajú všetky techniky okrem `iptables`, v prípade druhom dosahuje dvakrát vyššiu výkonnosť ako smerovací systém linuxového jadra.

Testovaním času inicializácie sa ukázalo, že XDP je schopné pokryť veľké rozsahy adries v rádoch milisekúnd. V prípade, že treba do jeho eBPF mapy pridať veľké množstvo pravidiel, škáluje pridávanie lineárne s počtom pravidiel na pridanie. XDP takisto vykázalo najkratší čas potrebný pre pridávanie pravidiel spomedzi ostestovaných techník.

Nakoniec zhodnocujeme, že implementované riešenie splnilo výkonnostné očakávania a zapodievame sa možnosťou rozšírenia o podporu IPv6.

Contents

Chapter 1

Introduction

Since the advent of inter-computer and inter-network communication and the invention of the Internet, we have been witnessing an ever-increasing load on our computer networking infrastructures, more so with recent surge in popularity of streaming services and the emergence of computer network attacks focused on machines connected to the Internet.

It is of high importance that the machines operating on the Internet are able to provide services under varying amounts of pressure and load, virtually without ever ceasing to function correctly. Every second that a company can not provide its services to the potential customer can lead to a significant loss of profit.

Many techniques have been therefore developed as means of hardening a system against potential threats from the network. In order to be effective, these defence mechanisms must be built with high performance requirements in mind.

In this work, we take a look at current practices in the field of network traffic filtering, we analyse the Berkeley Packet Filter and Express Data Path technologies with regards to their combined capability of packet filtering, and we propose a system for such filtering based on these technologies.

This work is divided into chapters as follows.

Chapter 2 describes current widespread practices of packet filtering solutions on Linux kernel based operating systems, then the Berkeley Packet Filter and the extended Berkeley Packet Filter (eBPF) are described as the packet filtering method of our interest. The chapter then defines the eXpress Data Path (XDP) as a technique implemented in network interface device drivers which allows the packet filtering to be performed as soon as an ingress packet is received on the hardware. At the end of the chapter, an XDP eBPF program example is provided and a method of loading it into the device driver is shown.

Chapter 3 proposes a system of filtering packets on the NETX router platform leveraging XDP and eBPF. It states a number of software requirements, such as how should a decision on a packet be performed, what eBPF data structures shall be used to store rules for packet filtering, or how its performance should compare to other contemporary packet filtering solutions.

In chapter 4, we shed light upon our implementation of the XDP filtering program. We describe what data structures are employed for rule storage, what the compilation process is for the restricted XDP programs, how to load the program onto a network interface and how to interact with it. A detailed description of two programs that comprise the filtering solution is provided here.

The implemented solution is then subjected to a series of performance tests whose results can be found in chapter 5. The chapter describes the methodology with which the tests were performed, what were the expectations regarding the tests, and the actual performance results. Four scenarios are presented, two regarding packet filtration and two regarding filtering system initialisation time.

Finally, a conclusion on the implementation and its performance is drawn in Chapter 6.

Chapter 2

Current State of Filtering Technology and Existing Solutions

The purpose of this chapter is to get acquainted with both packet filtering systems which have been used in the past and contemporary systems which may typically be seen in deployment today. Then, the Berkeley Packet Filter and Express Data Path mechanisms (which will be referred to as BPF and XDP, respectively) are analysed with regards to their practicality as techniques for packet filtering.

2.1 Mainstream Packet Filtering Solutions

The world of firewalling and packet filtering on systems based on the Linux kernel is dominated by `iptables` and `nftables` usage.

2.1.1 iptables

`iptables` is a user space tool for the manipulation of the Linux firewall which is implemented as a number of Netfilter [18] kernel modules. It utilises tables of the so-called chains as sequences of rules which are inspected sequentially. Generally, a rule contains a predicate against which a packed is compared, a verdict which is enforced if the predicate is true with respect to the processed packet, and an optional target which specifies a kernel module extension for more extensive packet processing.

Chains

There are five different chains which are inspected at different points of the networking stack. The `PREROUTING` chain targets all incoming packets before they are routed, the `INPUT` chain targets all packets that are destined to the system, the `FORWARD` chain is fired on packets which are being routed, the `OUTPUT` chain targets all packets originating from the system, and the `POSTROUTING` chain targets all packets outgoing from the system. A diagram of the chains and their interconnection with the operating system can be seen in Figure 2.1.

Tables

`iptables` contains the following tables.

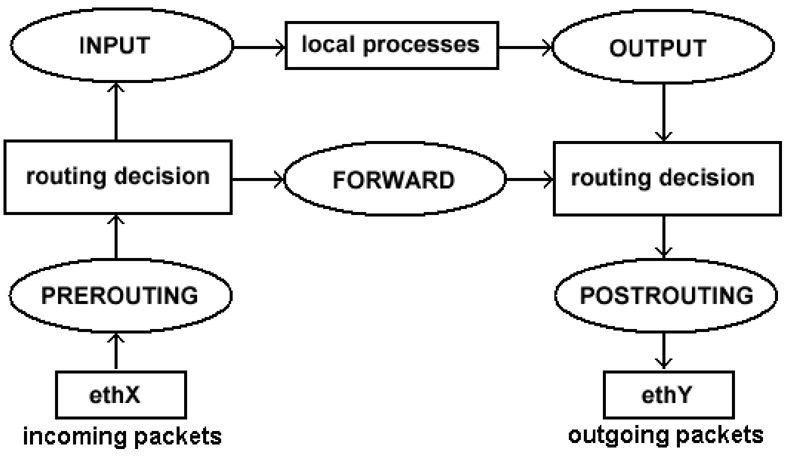

Figure 2.1: A diagram showing the order of `iptables`' chains and their interconnection with the operating and routing system. Diagram taken from [7].

The *filter* table is the default table and its purpose is to perform packet filtering on its `INPUT`, `OUTPUT` and `FORWARD` chains.

The *nat* table is consulted when the system encounters a packet that creates a new connection. It consists of the `PREROUTING`, `OUTPUT` and `POSTROUTING` chains [12].

The *mangle* table is used for packet alteration. Currently, this table has all five chains.

Finally, the *raw* table is used for creating exceptions from connection tracking and the *security* table can be used to enforce Mandatory Access Control networking rules.

ipset

A system administrator may often need to target ranges of IP addresses or ports instead of specific ones. A tool called `ipset` may be used in concert with `iptables` to create such ranges, drastically reducing the size of chains and increasing the speed of comparing an entry against a set of IP addresses [18].

2.1.2 nftables

The `nftables` is another Netfilter project introduced in Linux kernel version 3.13. It is the official replacement of `iptables` and its derivatives, presenting a packet filtering and processing system that is based on `iptables` (mainly by the use of rule chains).

Its main selling point is the performance increase with the help of a specialised, BPF-inspired virtual machine bytecode with a limited set of instructions. It also partly removes the linear complexity of walking the rule sets by aggregating the rules into maps which reduce the number of rule inspections [18].

2.2 The Berkeley Packet Filter (BPF)

As the technical product of this thesis relies heavily on exploiting the mechanisms which the Berkeley Packet Filter provides, the following sections provide an overview of the inner workings of its filtering mechanism.

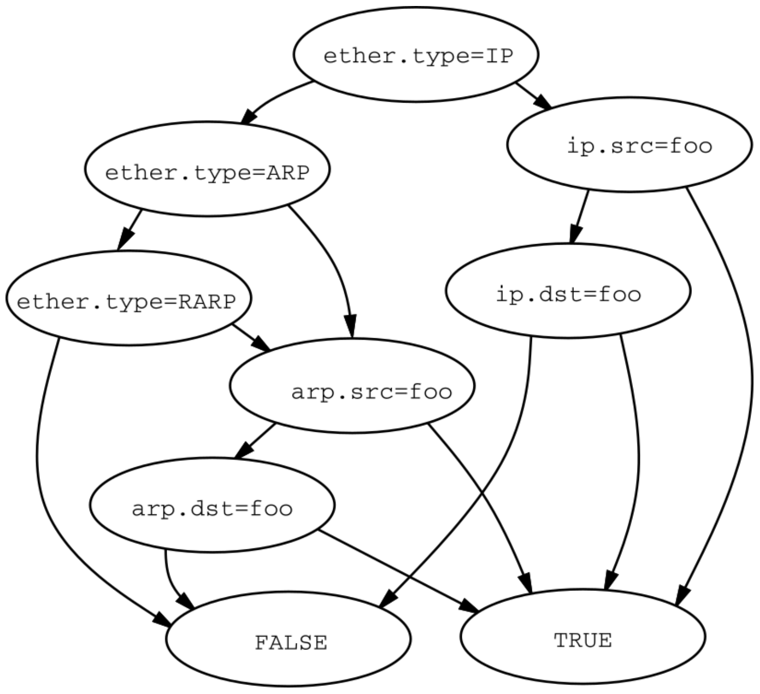

Figure 2.2: A CFG filter function that accepts packets from host *foo*. Taken from [15].

2.2.1 Origins of BPF

The Berkeley Packet Filter was originally introduced in a 1992 paper at the Lawrence Berkeley Laboratory as a system for filtering incoming packets as early as possible upon being captured by the receiving machine. This new filtering method had been developed with efficiency, extensibility and portability in mind; indeed, the filter had been up to 20 times faster than filtering mechanisms of the time and had been able to run on most BSD-based systems [15].

The introduction of BPF to the Linux kernel development tree has been first seen in version 2.5 [10].

2.2.2 The Filter Model

It is fair to say that a packet filter is a simple function on a packet, returning a boolean value. If the packet filter returns true, the packet is copied or forwarded to the kernel for further processing by the networking stack. If the packet filter returns false, it simply ignores the packet and drops it [15].

BPF uses a directed acyclic control flow graph (CFG) as its packet filter abstraction. In such graph, each node represents a predicate on a packet field while the edges, two outgoing for each node, represent the transfer of control to the next node. Finally, there are two leaf nodes representing true and false verdicts on the processed packet. Figure 2.2 illustrates a CFG that only accepts packets with an address *foo*, inspecting IP, ARP and RARP network protocol fields.

The CFG model has been preferred to a boolean expression tree, as the former model maps better into code for a register-based machine, while the latter into code for a stack-

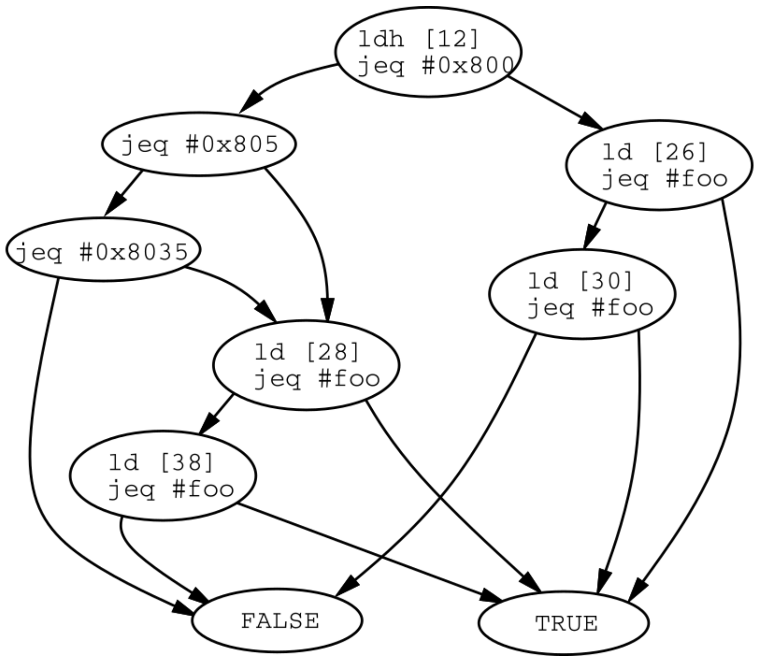

Figure 2.3: BPF program that accepts packets from host *foo*. Taken from [15].

based machine. Most computer systems operate as a register-based machine, therefore BPF is implemented as a CFG [15].

2.2.3 Virtual Machine

The implementation of the control flow graph model consists of two 32-bit registers, an accumulator and an index register which are mapped to physical registers, a scratch memory space, an array representing the packet and an implicit program counter, thus turning BPF into a virtual machine [15].

Various virtual instructions may be executed on the memory elements, such as load, store, logic, branching, return and miscellaneous instructions [15]. Most of the arithmetic operations are performed on the accumulator, while the index register provides offsets into the packet array or into the scratch memory space [9].

Figure 2.3 shows a control flow graph which is an adaptation of Figure 2.2 with virtual machine instructions instead of abstract predicates.

2.3 The Extended Berkeley Packet Filter (eBPF)

After its integration into the Linux kernel, BPF had remained relatively unchanged for much time until the 3.0 release of Linux with the addition of a just-in-time (JIT) compiler for the BPF interpreter [10].

Release 3.15 has brought an improvement of BPF, reflecting advancements in current hardware [13]. The new changes extend BPF by changing register width to 64 bits, increasing the number of registers from 2 to 10, being able to call a fixed set of in-kernel helper functions and by improving instruction execution performance, causing fewer cache misses [20][1].

Since the 3.15 version, the new BPF implementation has been called *extended BPF (eBPF)* and the original implementation *classic BPF (cBPF).*

2.3.1 The Just-in-time Compiler

Typically, instructions of eBPF are mapped 1:1 to respective assembly instructions of the system's hardware architecture. If it is supported by the kernel, the loaded eBPF program may be just-in-time (JIT) compiled into assembly code of the host system [19].

The simplicity of the eBPF virtual instruction set lends itself to an uncomplicated JIT translation. It maps every eBPF instruction to a straightforward sequence of x86 instructions and it uses the processor's registers as placeholders for eBPF's accumulator and index registers [9].

Early benchmarks of the JIT compiler have shown a 50 nanosecond save per invocation of a JITed eBPF program when compared with a program where such translation to machine code had not been employed [11].

2.3.2 The Static Verifier

As eBPF programs are run inside the kernel, certain precautions must be performed in order to preserve the security and stability of the kernel. Upon loading the eBPF program into the kernel, it must be subjected to a static verification.

First, the verification process checks that the program does not contain any loops, ensuring that it will not take a disproportionate amount of time to run, by executing a depth-first search on the program's control flow graph. Also, an eBPF program that contains unreachable instructions will cause the verifier to fail the analysis [13].

Second, the verifier simulates the execution of the eBPF program one instruction at a time, checking the virtual machine state after each instruction's execution. All jumps must land within the program and all memory accesses must not read or write outside the kernel-provided memory area. Registers and stack variables with uninitialised contents may not be accessed, as doing so would fail the verifier [13].

Finally, the verifier prohibits any user without administrative privileges to perform pointer arithmetic in an eBPF program they load. This is done to protect kernel addresses from unprivileged user access [13].

The static analysis does not need to walk through all possible paths of a program, as it performs path pruning based on comparing the current state to its history of accepted states [5].

2.3.3 The bpf() System Call

The `bpf()` system call can be used to perform a variety of operations on eBPF filters. It is defined as

```
int bpf(int cmd, union bpf_attr *attr, unsigned int size);
```

Listing 2.1: BPF system call declaration.

The `cmd` argument specifies which operation will be performed on the `attr` argument.

Two types of operations are supported. One operation for loading and verifying an eBPF program, `BPF_PROG_LOAD`, and several for eBPF maps creation and manipulation, such as `BPF_MAP_CREATE` to create a map or `BPF_MAP_LOOKUP_ELEM` to lookup an element in a map [1][5]. eBPF maps are explained in more detail in Section 2.3.4.

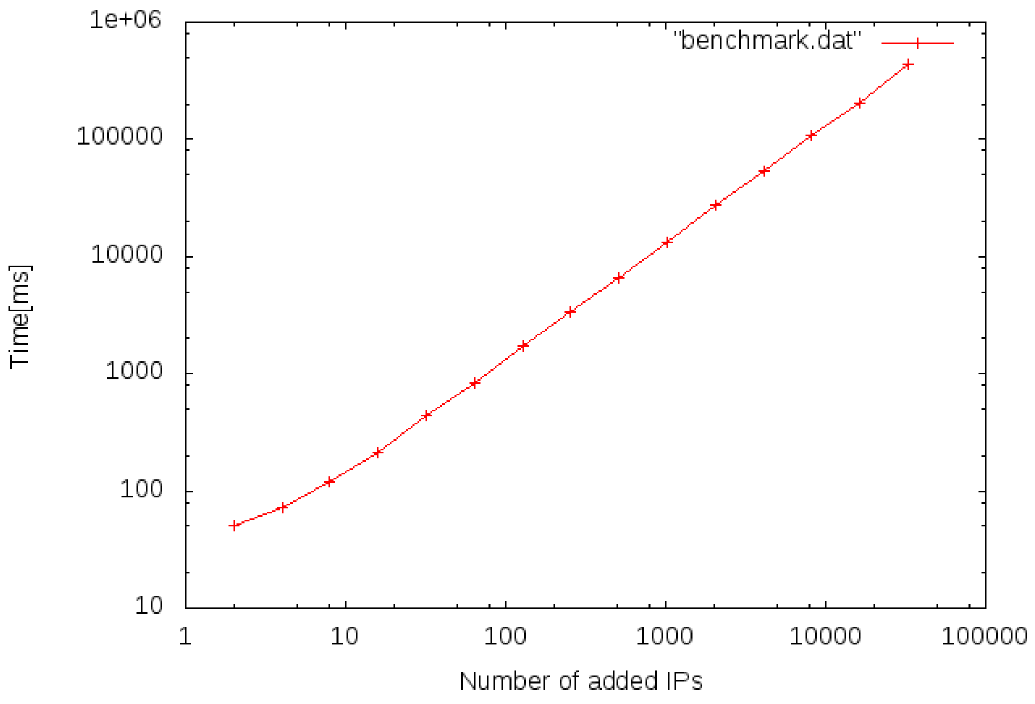

Figure 2.4: A graph showing the relation between the number of IP addresses to add to a hash map and the time it takes to add those addresses to the map. The sizes of the sets range from 2 addresses (a full /31 subnet; the shortest time), doubling each time up to 32, 768 addresses (a full /17 subnet; the longest time).

2.3.4 Maps

A map, in the context of eBPF, is a generic data structure that can hold different types of data for the purpose of sharing it between eBPF programs, and between kernel and user space programs [5]. The maps also provide a persistent storage of data between invocations of an eBPF program.

From a high-level perspective, a map hides each stored value behind a unique key. The means of accessing a specific value are implementation-specific, depending on used map type, such as an array or a hash map.

Using the `bpf()` system call mentioned in Section 2.3.3, a map can be created or deleted and a key-pair value of a map can be looked up, created, updated or deleted.

Many map types have been implemented to be used in eBPF programs. Several types are valuable for programs that implement packet filtering: arrays, hash maps and longest prefix match (LPM) tries.

The LPM tries are a very attractive option for a packet filtering program which selectively drops packets based on their source or destination IP address. A range of addresses can be stored in an LPM trie map with a single eBPF helper function, while the same range would need to be stored as separate addresses in an array map or a hash map. As it can be seen in Figure 2.4, the time it takes to add a set of IP addresses to a hash map grows linearly with the size of the set.

2.3.5 Program Types

When loading an eBPF program with `BPF_PROG_LOAD` (see Section 2.3.3), the type of the program limits the number of in-kernel helper functions available to the program. The program type also specifies where the program can be attached and it dictates the type of the object which is passed to the program as its first argument [13].

Among all the types of programs, two are relevant for a program which implements a packet filtering algorithm: `BPF_PROG_TYPE_SOCKET_FILTER` and `BPF_PROG_TYPE_XDP`. The value carrying the type of the program must be passed in the second parameter `attr` of the `bpf()` system call [13].

2.4 The eXpress Data Path (XDP)

Packet processing systems with a focus on high performance require strict constraints on the time spent processing each packet. Implementations of general purpose network stacks have lead to the creation of specialised systems for packet processing such as the Dataplane Development Kit (DPDK) because of their flexibility and thus inability to withstand high packet loads [14].

Such toolkits typically employ a kernel bypass technique. On one hand, a bypass can dramatically improve packet filtering performance. On the other hand, the bypass is more complicated to integrate with the operating system, as it cannot make use of functionality provided by the system, such as routing tables. Security risks may arise after such separation from the kernel, as the kernel can no longer enforce its security policies on the bypassing system [14].

Therefore, the eXpress Data Path (XDP) framework has been created as an alternative to such systems. It allows high speed packet processing while still letting the kernel enforce a safe execution environment. XDP works with concert with the extended Berkeley Packet Filter (eBPF) which provides a method of processing incoming packets before being touched by the kernel and at the earliest point after the packet is received from the hardware [14].

2.4.1 The Design of XDP

XDP is implemented as a hook in network device drivers immediately after receiving a packet from the hardware. This design comes with great performance advantages, as the eBPF program is run in the device driver without the need to switch context to the user space. The eBPF program is also allowed to modify the packet. Moreover, no socket buffer is allocated before the program is run, lowering the overhead even more, as the socket buffer allocation would be unnecessary if the processed packet is dropped.

The XDP hook can be seen in Figure 2.5 as the first action that the device driver performs. If the packet is destined to be dropped, the respective socket buffer memory does not need to be allocated. The XDP eBPF program may also decide to pass the packet to the networking stack for regular packet processing, to transmit it back onto the ingress network interface, to redirect it to another network interface, or to pass it to a user space application, bypassing the kernel processing with the use of the `AF_XDP` socket type. The `AF_XDP` is a novel feature of the Linux kernel, first support added in the 4.19 version [14].

2.4.2 XDP Actions

The ingress packet's processing path after the XDP hook is communicated from the eBPF program by specific return codes:

1. `XDP_DROP` causes the packet to be silently dropped without its data being copied to the kernel.

2. `XDP_ABORTED` signals an eBPF program error and should not be returned by any functional program. This action also causes the packet to be dropped.

3. `XDP_PASS` indicates that the packet shall be passed to the kernel for regular network stack processing. The eBPF program may have modified the packet before passing it to the kernel.

4. `XDP_TX` causes the packet to be sent back out of the ingress interface.

5. `XDP_REDIRECT` allows the packet to be redirected to and to be sent from another network interface.

2.4.3 Operation Modes

There exist three modes in which XDP can operate: the native mode, the offloaded mode, and the generic mode.

In the *native mode*, the eBPF program is run in the driver's early receive path. This is the default mode and it is supported by many widely-used NICs. This mode has been explained in Section 2.4.1.

The *offloaded mode* has the potential to be even faster than the native mode, as eBPF packet filtering programs are offloaded to the NIC to be executed at an earlier point than in the native mode. The eBPF program execution is in the hands of the NIC. This mode is only supported by SmartNICs (by Mellanox[1] or Netronome[2], for instance) which are equipped with multi-threaded processors and offer other network functionality offloading. These pieces of hardware also support the native mode of operation in case that some eBPF helper functions are not available [8].

Finally, the *generic mode* is offered by the kernel systems whose NIC device drivers do not implement the native or the offloaded mode of XDP. Since eBPF programs are run at a point in the networking stack, this mode operates at a slower rate and is therefore intended for development purposes [8].

2.5 XDP Example

This section shall present a minimal XDP eBPF program and demonstrate its compilation with the LLVM toolset and how to load it into the device driver.

The following is a minimal program which drops all incoming traffic on the interface:

[1]https://www.mellanox.com/related-docs/prod_adapter_cards/PB_BlueField_Smart_NIC.pdf
[2]https://www.netronome.com/products/smartnic/overview/

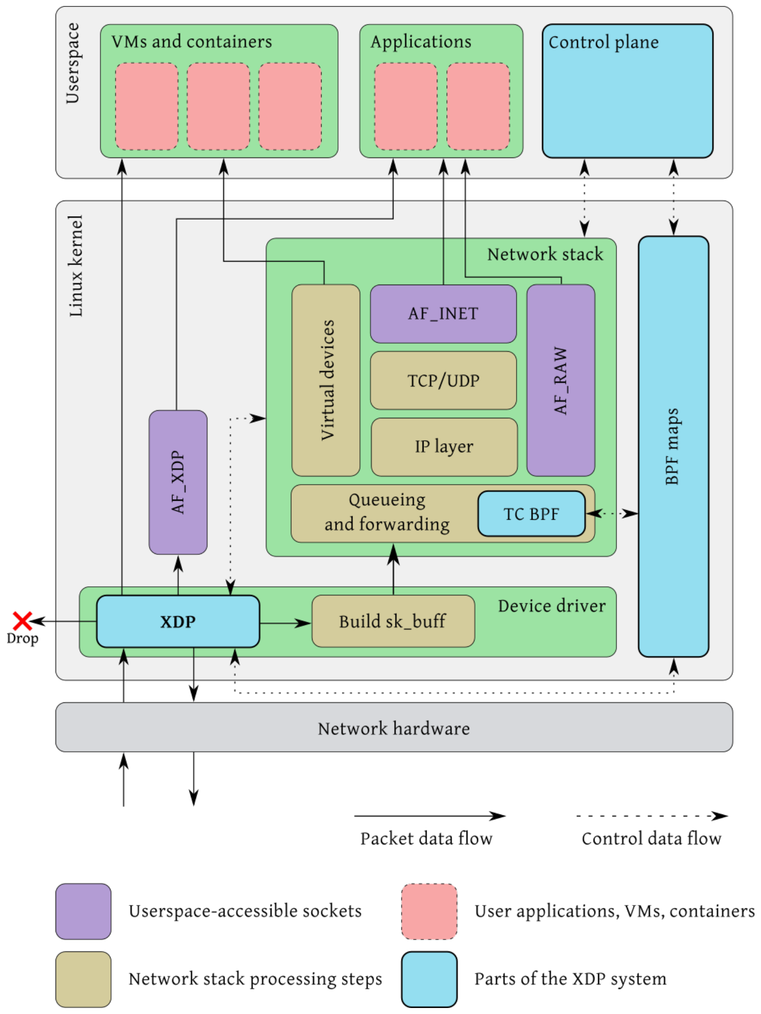

Figure 2.5: Design of the integration of XDP into the Linux kernel for ingress packets. Figure taken from [14].

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

#ifndef __section
# define __section(NAME) \
   __attribute__((section(NAME), used))
#endif

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return XDP_DROP;
}

char __license[] __section("license") = "GPL";
```

Listing 2.2: A minimal XDP program.

The program defines the `xdp_drop` function as the entry point using the `__section` macro. That macro causes the compiled program to contain a section called `prog` which is the default section name for an entry function. The entry function receives a pointer to the packet context, containing metadata about the ingress interface and pointers to the actual packet data.

Without accessing the packet data, the program immediately returns the `XDP_DROP` return value, indicating that the received packet will be dropped.

To compile this program, assuming that it is named `xdp-example.c`, the `clang` [3] tool may be used (version 3.9 and higher):

```
$ clang -O2 -Wall -target bpf -c xdp-example.c -o xdp-example.o
```

Finally, to load the program into the driver, the `ip` tool from the `iproute2` collection can be used. The following requires root access to the system, loading the `xdp-example.o` file into the device driver of the `eth` network interface:

```
# ip link set dev eth xdp obj xdp-example.o
```

2.6 Summary

In this chapter, we have shown in detail two Linux kernel technologies working in concert with each other which provide means of high-speed packet processing at the earliest point after receiving packets from networking hardware.

The extended Berkeley Packet Filter (eBPF) provides a specialised environment for packet processing, utilising a virtual machine with a specifically tailored instructions set architecture for accessing and modifying packet data and performing decisions based on packet fields. This technology is an extension of the Berkeley Packet Filter (BPF) – the just-in-time compiler, maps as generic data stores and the static verifier are additions to BPF which allow the creation of more flexible and computationally safe packet filtering programs.

The eXpress Data Path is more secure kernel bypass technique alternative which is more secure and works more cooperatively with the kernel with the intention of providing a very early point of packet processing and a method of bypassing the kernel. It is implemented as a hook in network interface device drivers where the eBPF program can be run before any packet data is copied to the kernel.

Chapter 3

Design of the Proposed System

Based on the knowledge contained in Chapter 2, a design of a packet filtering program on a specialised routing system is proposed. The target routing system called *NETX* is therefore first described. The filtering program serves as a method of DDoS[1] protection for this routing system.

The target platform description is followed by an enumeration of functional requirements of the proposed system, including how to operate it, what tasks it should be able to perform and what computer environment constraints must be fulfilled in order for the system to function properly.

Finally, we describe how XDP and eBPF shall be utilised for maximum efficiency of the proposed system.

3.1 NETX

NETX is an open-source routing platform developed at Brno University of Technology with focus on high routing performance and the provision of a rich set of routing features [17]. NETX routers are designed to handle multiple BGP tables, they support a wide variety of networking protocols and they are capable of routing performance of 60 Gbps [16].

NETX's operating system is based on GNU/Linux, allowing easy extensibility and adaptability to various networking tasks. Moreover, the NETX system features a robust configuration API.

3.2 Software Requirements

Certain requirements for the filtering system have been set in order to maximise the system's utility and performance.

3.2.1 Packet Decision

The proposed system shall perform as a highly specialised filtering program. It shall process all incoming packets on a network interface, inspect the appropriate header fields and perform an action on the packet based on the header fields and a set of rules. If an

[1]Distributed denial of service attack is a network-based attack on a computer system where the victim system is flooded with incoming packets coming from many sources, causing the system to generate very high amounts of interrupt requests which subsequently cause it to halt and be unresponsive to normal service requests.

incoming packet matches a filtering rule, the verdict associated with that rule is applied to the packet. The verdict shall be specified as one of the following XDP actions: pass, drop and redirect. If the packet matches no rules, the implicit rule is enforced – to redirect the packet to the egress network interface.

The mentioned rule set can therefore also be called a *whitelist*, marking the matching packets as packets that should not be implicitly redirected, although in the end may be redirected as an effect of a rule's verdict.

Unless the packet data is corrupted or an error occurs while parsing an incoming packet, no packet shall be dropped unless there is a rule that specifies that the matching packets shall be dropped.

In the remainder of the text, "packet redirection", "packet forwarding", "redirection", or "forwarding" shall describe the passing of a packet from ingress network interface to egress network interface without modifying the packet's contents.

3.2.2 Rule Set Storage

The rules against which the incoming packet's fields would be compared shall be stored in a data store that is easily accessible to the filtering program. As XDP programs are executed separately for each incoming packet and as filtering rules may be changed at any point in time, there must exist a separation between the XDP program and the data store for filtering rules. A straightforward solution is provided by eBPF maps, as mentioned in Section 2.3.4, whose generic structure can be used to store arbitrary data such as the filtering rules.

Adding to or deleting from the rule set should not interfere with the filtering process in place, except to the extent of the modified rule. A modification of the rule set should cause the program to continue filtering seamlessly. However, the program may be unloaded from the network interface, and the entity responsible for the unloading is also responsible for clearing the rule set data store, so that when the filtering program is loaded on the network interface again later, it starts its computation with an empty rule set.

3.2.3 Performance Metrics

With regards to processing speed and CPU intensity, the filtering system should display marginally better results than three other approaches to packet forwarding:

1. forwarding with iptables,
2. forwarding with iptables and ipset,
3. Linux routing system forwarding.

These approaches shall be compared based on the following metrics:

1. processing speed in packets per second,
2. time it takes to load filtering rules into memory,
3. CPU load.

Results of the performance testing can be found in Chapter 5.

3.3 eBPF and XDP Involvement

The extended Berkeley Packet Filter (eBPF) and eXpress Data Path (XDP) technologies shall be utilised for the packet filtering solution.

XDP shall be used as a technique which allows ingress packet analysis at the earliest possible point in network interface device drivers, redirecting traffic immediately without the added, unnecessary overhead of copying the packets' data to the kernel.

In concert with XDP, eBPF will provide an environment for high-speed packet analysis. The eBPF maps will be utilised for storing packet filtering rules. The longest prefix match trie eBPF map type shall be employed for this purpose, allowing ranges of IP (or IPv6) addresses to be stored and single addresses to be checked for a match.

There will be three possible outcomes of processing an ingress packet. The packet can either be dropped if its data is erroneous or an error occurred during its processing, or it can be passed to the kernel for normal network stack processing, or it can be redirected to an outbound networking interface for further analysis by a computer system residing at the receiving end of the outbound interface. These scenarios correspond with the `XDP_ABORTED`, `XDP_PASS` and `XDP_REDIRECT` XDP actions, respectively.

Chapter 4

Implementation Details

This chapter aims to provide a detailed description of the filtering system that has been implemented as the centrepiece of this thesis.

The following sections shall describe the code base of the filtering system, the compilation process from the source files to the binaries, how the program is loaded into the network interface driver, and what is the command line interface of the loading program.

Finally, a step-by-step analysis of the filtering XDP program is shown with explanations of each action it performs.

As mentioned in the following section, the packet filtering implementation consists of two important programs. One of them is the program that is loaded into the network interface's driver and performs packet filtering; This program shall be referred to in this chapter as the "XDP program", "(e)BPF program" or "filtering program". The other one is responsible for loading and unloading the eBPF program from the driver, and adding and removing filtering rules from the packet filter; This program shall be referred to as the "XDP program loader", "(e)BPF program loader", or simply the "loader program"[1].

4.1 Code Base Layout

The source code is an adaptation and an expansion of the XDP Tutorial[2] which was revealed and presented at the 2019 NetDev conference[3]. It is completely written in C (and partially in restricted-C) as a constraint placed upon the implementation by the available compilers but largely by the BPF library which is implemented in C.

The XDP program must follow the restrictions of its programming logic as dictated by the constraints of eBPF's static verifier in Section 2.3.2, e.g., no loops, no out-of-bounds memory accesses, and others, hence the restricted-C usage. No such constraints are placed on the loader program which is also written in C.

The source code is divided into four directories:

- **`src`** contains the core logic of the packet filter – the XDP program and the loader program. A close look at those programs can be found in Section 4.6 and 4.4, respectively.

[1]The reasoning behind the naming is that it increases clarity in conveying information, as using the actual file names could hinder it.

[2]`https://github.com/xdp-project/xdp-tutorial`

[3]`https://www.netdevconf.org/0x13/session.html?tutorial-XDP-hands-on`

- `common` contains either header files that are common to the XDP and loader programs, or files that do not represent the filter's core logic, such as command line argument parsing.
- `headers` contains header files containing either BPF forward function declarations or definitions of macros for the XDP program.
- `libbpf` contains the source code of the libbpf library. The library is utilised to ease development and to load eBPF programs.

4.2 eBPF Maps as the Rule Set

As mentioned in section 3.2.2, the filtering solution uses eBPF maps to store rules. It performs a filtering decision in each XDP program invocation (one packet per invocation) on the basis of these rules. Our solution uses two maps.

The first map, called `tx_port` stores information about which network interface is the egress interface. The map is of the `BPF_MAP_TYPE_DEVMAP` type, meaning its keys and values are both integers. This map stores only one value, and that is the egress interface number under key zero.

The second map, called `lpm_whitelist` is the filtering rule map which stores ranges of IP addresses. The map is of the `BPF_MAP_TYPE_LPM_TRIE`, meaning that its entries are stored in a longest prefix match (LPM) trie, a data structure that allows fast LPM lookups and which is typically employed in routing tables to store network routes. Its keys are ranges of IP addresses specified as a tuple of a network address and the prefix length. Values of the map's entries are XDP verdicts, i. e., redirect, pass, drop or others. In the implementation, only the pass verdict is stored in the map, as the filter's logic is to redirect all traffic unless the destination address matches an entry in the trie map, in which case the verdict is to pass the packet for further processing by the kernel.

4.3 Compilation Process

There are not many available compilers that have the capability to compile a restricted-C file into eBPF byte code stored into an ELF object file. Fortunately, the LLVM compiler infrastructure contains two compilers which can work together to deliver the desired binary file. Those two compilers are *clang* [3] and *llc* [6]. The lowest LLVM version required for the compilation is 3.9 [8].

The compilation of the XDP program's source code into a binary file is a two-step process, requiring an intermediary compilation into an LLVM intermediate language assembly file which is, as its name suggests, a file containing the filter's instructions in an intermediary format (eBPF bytecode) between compilations of the LLVM toolkit.

This first compilation is performed by the *clang* compiler. The program's options required to create the intermediary file are `-S -target bpf -emit-llvm`. In the order of mentioning, these options commend *clang* to stop the compilation pipeline before producing a binary file, to target the BPF architecture (in another words, to create eBPF byte code), and to generate an LLVM intermediate language assembly file. By a naming convention, the intermediary file has the `.ll` suffix.

The LLVM file containing the eBPF instructions is then compiled into an ELF object file with the *llc* compiler. In order to create the binary file, this program must be supplied, among others, the following options: `-march=bpf -filetype=obj`. These options commend *llc* to target the BPF architecture on input, and to emit an ELF object file. By convention, a file with the `.o` suffix is generated.

This ELF object file represents the loadable XDP program.

4.4 XDP Program Loader

This section shall describe what is the functionality of the XDP program loader and how it is achieved. There exist two methods of loading an XDP program onto a network interface (also loading it into the kernel) on a Linux machine, either using the `ip` tool from the `iproute2` package, or making use of the *libbpf* library.

The first method is shown with the intent of showing how the `ip` tool can be used to load the XDP program, however this technique is not employed in the final filtering solution.

Then we show the second method which has been chosen for the solution, using the *libbpf* library. In the description of this method, we show how the library is leveraged to not only load the XDP program but also to update the rule set and unload the XDP program.

4.4.1 The `iproute2` Method

The first method involves using the `ip` utility from the `iproute2` package, as mentioned in Section 2.5, by invoking it as follows. Let `eth` be the network interface onto which the XDP program would be loaded, `xdp-example.o` be the XDP program. By default, the utility searches for the `xdp_prog` section in the XDP program as the entry point of the program. Running this utility with the purpose of loading an XDP program requires running it with superuser privileges.

```
# ip link set dev eth xdp obj xdp-example.o
```

There is one downside to this method; It does not support creating and utilising eBPF maps because the loader is not based on the libbpf library which does support eBPF maps. The design of our program requires a persistent data store between the user space and the kernel, as mentioned in Section 3.2.2, which renders this method insufficient.

4.4.2 The BPF Syscall Method

The second approach leverages the `bpf` syscall to perform a variety of actions, including loading and unloading the XDP program onto and from a network interface, or updating the eBPF maps with rules.

The key component here is the *libbpf*[4] library which eases the development of eBPF programs by providing wrapper functions for the syscall and helper functions for the restricted XDP programs, e.g., to perform eBPF map lookup. The library resides in the top directory of the source code and must be compiled and installed onto the system before its functions can be invoked.

All mentioned function names belong to the *libbpf* library unless stated otherwise.

[4] https://github.com/libbpf/libbpf/

Loading the XDP Program

Excluding the command line arguments parsing, the loading of an XDP program onto a network interface consists of the following steps.

- Load the eBPF ELF file into the kernel. All XDP programs in the supplied XDP binary are loaded into the kernel calling the `bpf_prog_load_xattr` function. In turn, the kernel evaluates the programs with the static eBPF verifier.

- Find a matching eBPF program section name. As multiple programs may be supplied with one XDP binary, in this step the program whose name matches the user-supplied name is selected calling the `bpf_object__find_program_by_title` function.

- Get the selected program's file descriptor. The file descriptor, represented by an integer number, is required in the next step. It is received by calling the `bpf_program__fd` function.

- Attach the XDP program described by the file descriptor to the supplied network interface driver's XDP hook. For this purpose, the `bpf_set_link_xdp_fd` function is called with appropriate flags, specifying whether the XDP program should be run in the native mode (the `XDP_FLAGS_DRV_MODE` flag) or in the generic mode (the `XDP_FLAGS_SKB_MODE` flag), or in the offloaded mode (the `XDP_FLAGS_HW_MODE` flag).

- Update the output network interface map with the supplied egress interface number. First, the `bpf_object__find_map_fd_by_name` is called, supplying it with the object returned by the library call in the second step and the map name. This call returns the file descriptor of the map. Second, the file descriptor is supplied as one of the parameters of the `bpf_map_update_elem` function, along with the key (set to zero[5]) and the number of the network interface.

- Pin the ebpf maps. This step ensures that the ebpf maps are available for updating even after the loader program finishes loading the xdp program. The maps must be pinned because the loader program must be run again (albeit with different parameters) to update the contents of the maps – to update the filtering rule set. The maps are pinned in the `/sys/fs/bpf/<if-name>` directory. Placing them in the directory named after the interface prevents potential collisions in map file names if the filtering program is loaded concurrently on two network interfaces in the same system. The `bpf_object__pin_maps` is called and supplied the object returned from the second step and the path to the target directory as a string.

Adding a Filtering Rule

Adding a rule to the rule set map is a fairly simple process. The loader program must be supplied the name of the interface on which an XDP program has been loaded, and the IP network address with a prefix length.

The loader first converts the IP address from a string into an internal representation compatible with the *libbpf* library. Then, it tries to open the pinned rule set map file by calling the `bpf_obj_get` function, passing it the path of the map file. If successful, it uses

[5]The reason for this is that the map shall always contain only one output interface.

the returned file descriptor as a parameter of the `bpf_map_update_elem` function along with the key (the IP address and prefix length structure) and the stored value (`XDP_PASS`).

If the map file cannot be opened, or if the update function fails to add the new rule to the rule set, the loader program reports the erroneous behaviour as a message on the standard error file descriptor.

The update function (more precisely, the BPF syscall command logic) does not distinguish the successful entry addition where there had been no previous value from a successful entry addition where there had been a value on that particular key. Therefore, the user is only notified of errors when adding a rule.

Removing a Filtering Rule

The deletion of a rule from the rule set mirrors the rule addition behaviour almost identically.

The loader program must be supplied the IP address, the prefix length and the network interface's name on which to operate.

After the loader converts the input data into the internal representation, it calls the `bpf_map_delete_elem` function. The function, in comparison with the update function, expects only the rule set map file descriptor and the IP (and prefix length) key as its parameters.

The loader reports erroneous behaviour on the standard error file descriptor, such as trying to delete from a non-existent map, trying to delete a non-existent entry or supplying an invalid IP address.

Unloading the XDP Program

Before the XDP program can be unloaded from a network interface, all maps associated with this interface must be unpinned and deleted.

Therefore, the loader program must first load the BPF ELF file into memory calling the `bpf_prog_load_xattr` function. Do note that this step is the same as the first step of the loading procedure; The binary file is loaded because it contains information about maps that are associated with the XDP program and because the structure pointer that is returned by the function call must be supplied to the following *libbpf* function call.

The loader then tests if one of the map files exists by calling the `access` function of the POSIX operating system API (by including the `unistd.h` header file). If the test access was successful, the `bpf_object__unpin_maps` function is invoked, supplying it the BPF file structure pointer from the previous step and a string containing the path to the directory of the pinned maps.

4.5 Loader Command Line Interface

It can be seen that with regards to functionality, the XDP loader is no simple program. Therefore, an appropriate command line interface has been developed for the loader.

The command line interface consists of a number of options that can be passed to the loader. What follows is a list of the options accompanied with their descriptions:

- `-h`, `--help` prints the synopsis of the program to the standard output.
- `--in <if-name>` specifies on which input interface should one of the commands operate. The network interface must be specified by its name.

- `--out <if-name>` is both a command to load an XDP program onto the input network interface and a specification of the interface to which incoming packets shall be redirected.
- `--unload` commands the loader to unload the XDP program from the input network interface specified with `--in`. It should be accompanied with the same XDP mode option which was supplied when loading the XDP program.
- `-a`, `--add <IP>/<prefixlen>` commands the loader to add the IP and mask to the rule set map associated with the `--in` interface.
- `-d`, `--del <IP>/<prefixlen>` commands the loader to remove the IP and mask from the rule set map associated with the `--in` interface.
- `-v`, `--verdict <action>` specifies what action shall be taken if an incoming packet matches the rule added with `--add`.
- `-S`, `--skb-mode` specifies that the XDP program should be operating in the generic mode. Beware that this mode drastically reduces the performance of the packet filter.
- `-N`, `--native-mode` specifies that the XDP program should be operating in the native mode.
- `-A`, `--auto-mode` lets the loader program decide the mode of operation by itself. It first tries to load the XDP program in the native mode. If that mode is not supported by the network interface's driver, then it tries loading the XDP program in the generic mode of operation.
- `--offload-mode` specifies that the XDP program should be operating in the offloaded mode.
- `-F`, `--force` can be used in combination with the `--out` option and it directs the loader to unload any existing XDP programs from the input interface when loading an XDP program.
- `-q`, `--quiet` causes the loader to be less verbose in its output.
- `--filename <file>` causes a specific BPF ELF file to be loaded as the XDP program instead of the default one (`xdp_prog_kern.o`).
- `--progsec <section>` causes a specific section of the BPF ELF file to be chosen as the XDP program, instead of the default one (`xdp_redirect`).

It should be noted that not all combinations of options are valid, i.e., no two "command" options can be specified simultaneously (`--out`, `--unload`, `--add`, `--del`) in which case the loader program will halt with an error message.

The mode of operation flags should not be mixed. This behaviour is however not enforced, the mode flag supplied as the last one on the command line is taken into account.

Most of the options are optional; The loader program supplies sane default values to unspecified parameters or it performs a default procedure (for example, it tries to load the XDP program in the native mode if no mode of operation is specified). The input interface option is a required one, as well as specifying a command is required.

Finally, some options are ignored, such as `--force` or a mode of operation specification when adding or deleting a rule.

4.6 XDP Filter Program

The XDP filter program is the core of the filtering solution as described in Chapter 3. This section shall detail the implementation of the program.

As stated before, the computing environment restricts the implementation language capabilities of the program. Although it is written in C, it lacks certain attributes of the language. No backward jumps are allowed (although loops with a predetermined number of iterations can be unrolled with the `#pragma unroll` preprocessor directive), jumps may not be performed to user-supplied memory addresses, and all data accesses must be preceded with a certain check for out-of-memory bounds.

The implemented XDP program consists of three main parts: declarations of eBPF maps, inline helper functions, and an entry function. Of course, standard include directives are present, as well as helper structure declarations.

The maps and the entry function must be placed in specific sections of the ELF binary. Therefore, the built-in `__attribute__` can be specified in the declaration of the appropriate objects, supplying it with the name of the target ELF section. The maps must be placed in the `maps` section of the binary, and the entry point must be assigned a section name that is different from the name of the C function that represents the entry point. The entry function (the main XDP program) can be then referenced by the section name which is in the implementation's case `xdp_redirect`.

4.6.1 Maps

The maps are declared as instances of *libbpf*'s `struct map_type_def`. The first eBPF map is the egress network interface map. It is declared as a `BPF_MAP_TYPE_DEVMAP` map type and at any given time it holds only one entry at key zero and that entry is the egress interface number. The map must be updated with the network interface number by the loader program when the XDP program is loaded into the kernel.

The second eBPF map contains filtering rule entries with IP-prefix keys and verdict values. It is declared as a map of the `BPF_MAP_TYPE_LPM_TRIE` type whose key size must be declared as the size of *libbpf*'s `struct bpf_lpm_trie_key`. As the declaration of the struct specifies an array of an unknown number of bytes for storing the IP address, the size must be calculated manually (as opposed to using the `sizeof` built-in).

More details about the maps can be found in Section 4.2.

4.6.2 The Entry Function

Every XDP program must be declared as a function that accepts a pointer to the `struct xdp_md` metadata structure and returns an integer. The implemented XDP redirect program follows this interface and extracts two values from the metadata context, namely the pointers to the beginning of the packet data and to the end of it. These shall be called the data pointer and the data end pointer, respectively.

These data pointers must be used to check ingress packet memory bounds before accessing any data in the packet, as the static verifier does not allow packet memory accesses that had not been previously checked. The memory check works as follows:

1. Calculate what is the maximum data offset (in bytes) of the desired data from the beginning of the packet data or from a valid pointer to the data (including an already checked offset).

2. Add the offset value to the data pointer.
3. If the new value plus one is greater than the data end pointer, abort. This means that a data access could possibly access data that is outside of its valid memory bounds. Continue with the packet processing within the checked offset otherwise.

At the beginning of the entry function, the data pointer is checked against the data end pointer with zero offset. This ensures that the packet data is valid, otherwise the packet is dropped with the `XDP_ABORTED` return code.

The program then parses the Ethernet header which is present at the very beginning of the packet data. The header parsing extracts the layer 3 header version and the offset of the next header in the packet data. Before the Ethernet header data can be accessed, the data pointer must be checked against the data end pointer with the Ethernet header size offset. The Ethernet header structure is included from the `linux/if_ether.h` header file.

Before the IP version extraction can be performed, the possibility of the packet being VLAN-tagged must be considered. This can be confirmed by looking at the `h_proto` field of the Ethernet header struct. If the packet is tagged, then the VLAN tag is inspected (after checking the appropriate memory) for the IP version value. If, by any chance, the packet is VLAN-tagged multiple times, this process is repeated until the packet is no longer tagged or a limit of VLAN encapsulation is reached. Finally, the IP version is extracted from the appropriate header field, and the data pointer is moved to the beginning of the next header.

A similar process is performed with the layer 3 header. The part of the packet data where the header resides is checked with the appropriate header's size as the offset. Then, the packet data at the data pointer is cast to the appropriate header struct.

The source IP address is then extracted from the header to the rule set map's key struct. This struct is then supplied to the `bpf_map_lookup_elem` call, looking up the source address in the longest prefix match trie. If the lookup ends positively, the returned value replaces the default verdict on the packet.

Finally, if the packet's verdict is to redirect it, the `bpf_redirect_map` function is invoked, passing it a pointer to the outbound network interface map and a key that points to the only network interface number in the map. This causes the packet to be marked for redirection to the appropriate egress network interface. The default verdict value is otherwise returned.

Chapter 5

Performance Testing

In order to understand how effective the filtering solution is, a set of performance tests has been executed. This chapter dives into the methodology of the benchmarks, what computing environment has been used to conduct the tests, what were the expectations regarding the implementation and what results have the tests achieved.

5.1 Methodology

The testing environment consisted of three Linux machines. The first one was responsible for generating dummy traffic (the generator machine), the second one was the machine on which the XDP program was loaded (the filter machine), and the third one was target of packet redirection (the target machine). The second and the third machines were running the NETX operating system (described in Section 3.1).

In general, the tests were executed as follows. First, the filter machine performed certain steps to remove any filters from a possible previous test scenario. Then, it applied filters to the system dictated by the current test scenario. The generator machine started in turn generating traffic at the highest possible rate towards the filter machine with parameters also dictated by the scenario. For the duration of the test, either the filter machine or the target machine were used to collect measurements of the test, e.g., the number of packets dropped or redirected per second. In some cases, the generator was not employed at all. In those cases, the filter machine was tasked to measure the time complexity of various filter initialisations. Finally, after a certain amount of time, the generator machine stopped generating traffic and the test measurements were collected from the appropriate machine.

The packet rate measurements were performed by repeatedly reading the packet counters of the appropriate network interface from the `/sys/class/net/<device>/statistics` folder's `rx_packets` and `tx_packets` files. The actual packet rate was then calculated as the difference of the counter's measurements one second apart. The resulting packet rate was then calculated as the average of those readings.

Other methods of packet rate measurements were considered, such as reading the above mentioned statistics of the target machine's receiving network interface or writing an XDP program that captures and records all incoming traffic on the receiving interface. However, due to the nature of some of the filtering scenarios, the redirected traffic could not be measured those ways on the interface, as the generated packets were redirected with their original L2 address which caused them to be dropped by the interface even before reaching the XDP program or being counted in the interface's statistics.

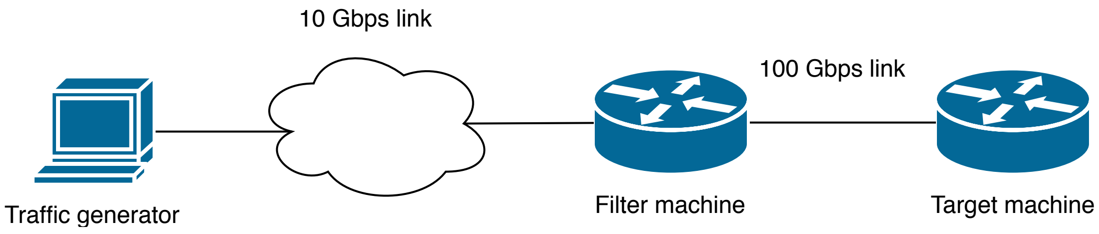

Figure 5.1: The networking topology that was used in all testing scenarios.

5.1.1 Tested Filtering Technologies

In addition to XDP, three other approaches were tested for performance. The `iptables` and `ipset` tools were used in scenarios where packets were dropped by the filter machine, and the state-of-the-art Linux kernel routing system approach was used in both packet-dropping and packet-redirecting scenarios.

On one hand, when filtering packets coming from ranges of IP addresses, the `iptables` approach required adding every address of the range as a single rule. On the other hand, specifying the same range (e.g., subnet) with the help of `ipset` required marginally fewer `iptables` rules. In many testing scenarios the number of such rules was one. The same number describes the amount of routing table entries required to test the Linux routing system approach.

5.1.2 Networking Specification

The performance tests used the following network topology.

As can be seen in Figure 5.1, the generator machine was connected to the filter machine with a single ten gigabit link. A switch lied on this link, however it was able to forward the traffic from the generator machine to the filter machine at any generated rate.

The generated traffic could possibly be (depending on the scenario) redirected to a direct link between the filter machine and the target machine. This was a link capable of carrying traffic of one hundred gigabits per second. This would have been the link onto which test traffic would be generated, however the interfaces' drivers did not support our preferred packet generating solutions.

Allocation of IP addresses in the networks between the machines was mostly irrelevant for the purposes of most scenarios. Although the source IP address of the generated packets was always inspected by the filtering machine, delivery of those packets between the two pairs of machines[1] was not dependent on it.

5.1.3 Test Machines Specification

The generator machine was running Red Hat Enterprise Linux (RHEL) version 7.2 with Linux kernel version 3.10. No modifications were done to the machine except that the outbound interface's driver was substituted with one that allowed line-rate traffic generation.

The filter machine was a NETX router based on RHEL 7.5. The supplied kernel version was not high enough to support the majority of XDP features and it was updated to kernel version 4.20 which was sufficient for all used XDP and eBPF features. The machine's

[1]As in generator-filter and filter-target.

processor was an Intel Xeon D-1587 with 16 cores (32 with multithreading) at 1.70GHz (2.30GHz maximum turbo frequency) and 16 GB of RAM.

Finally, the target was also a NETX router based on RHEL 7.5 with an Intel Xeon D-1537 CPU with 8 cores (16 with multithreading) and 16 GB RAM. There were no performance or feature constraints placed on the machine, so it was left with its default 3.10 kernel.

5.1.4 Data Generation

It is a common practice to benchmark networking-related applications with the smallest Ethernet frame size of 64 bytes. With this frame size, the ten gigabit link can potentially transfer about 16 million of such packets per second. However, Ethernet frames are preceded with a 12 byte inter-frame gap and an 8 byte MAC preamble which effectively pushes the minimum Ethernet frame size to 84 bytes [2].

The packet rate of the 84 bytes frames on a ten gigabit per second link can be calculated as

$$r = \frac{10 \cdot 10^9 \ \frac{bits}{s}}{84 \cdot 8 \ bits} \tag{5.1}$$

which equals to

$$r = 14,880,952 \ \frac{packets}{s} \tag{5.2}$$

The generator machine utilised the PF_RING project[2], a high speed packet capture and generation library implemented as a kernel module. The library's `pfsend` program was used as the packet generation program for all performed tests and it also supplied a driver for the outbound interface which was loaded in the kernel for the duration of the performance tests.

To provide an example, the following pfsend invocation generates traffic on the `enp1s0f1` interface (the `zc:` prefix signifies that the library's zero copy functionality shall be used), supplying packets from the `test-64.pcap` file, rewriting the generated packets' destination MAC address to that of the filter machine's inbound network interface (the `-m` option). It shall stop generating after sending ninety million packets in total (`-n 90000000`). The source and destination IPv4 addresses shall be rewritten with a range of 254 addresses (`-b 254`), starting with the appropriate destination (`-D`) and source (`-S`) addresses, simulating communication between hosts of two subnets of prefix length 24. The traffic shall be generated at ten gigabits per second (`-r 10`).

```
pfsend -i zc:enp1s0f1 -f test-64.pcap -m ac:1f:6b:2c:9e:71 \
    -n 90000000 -b 254 -D 100.91.0.1 -S 10.0.0.1 -r 10
```

5.2 Expected Results

The expectations about the tested technologies' performances were prior to the testing as follows.

With regards to packet dropping, the XDP program was expected to display the best per-core performance, as it inspects the incoming packets at an earlier point than the other technologies do. Then, we expected the Linux kernel routing system and the `ipset` technique to display similar performance, as they both operate in the kernel (the latter

[2] https://github.com/ntop/PF_RING

via a kernel module) and employ an IP range approach to perform filtering decisions on incoming packets. Finally, the `ipset` approach was expected to perform the worst, as it must contain one rule per IP address, making filtering large IP ranges difficult because of the sequential nature of its rule matching mechanism.

With regards to packet redirection, the XDP program was again expected to display the best performance for the same reason as in the packet dropping scenario. The `iptables` and `ipset` approaches could not be utilised for redirection, as they did not support it. The Linux kernel routing system was expected to have a worse performance than the XDP program.

5.3 Test Results

Based on the testing methodology and the environment described in Section 5.1, our solution's performance was examined in the following scenarios.

In all scenarios, the generator machine generated 64 byte long packets at 10 gigabits per second, or 14.48 packets per second.

5.3.1 Packet Dropping

In this scenario, the filtering approaches were configured to drop packets coming from a certain IP network. The size of the source IP space ranged from a /24 network (256 hosts) to a /18 network (16384 hosts[3]).

The approaches are first compared in filtering IP ranges and then in a setting with only one source IP, comparing them on a per CPU core basis.

As can be seen in Figure 5.2, the XDP filter, the `ipset` technique and Linux routing achieved dropped packet rates of about 11 million packets per second, while the `iptables` solution performed worse with each increase of the source network because of the linear increase of its filtering rules and the sequential nature of the rule comparison.

The dropped packet rate performance of the former three approaches did not weaken because the number of entries required to filter the generated traffic could be kept at a constant number. Also, the approaches, with the exception of `iptables`, did not need to utilise all CPU cores to 100%; The XDP program did not use more than 5% of total processing power while the ipset and Linux routing approaches used about 15% of total processing power.

Although none of the approaches achieved the potential maximum packet rate (displayed as the horizontal black bar at 14.88 Mpps at the top of Figure 5.2), the rate they achieved seemed to be a recurring maximum rate for the system on which the tests were carried out. Most of the differences can be explained by the need to exclusively access shared data structures and by a possible failure of the processor's DDIO facility [4] that places packet data into the L3 cache [14].

Per CPU Core Performance

The packet dropping scenario was then adjusted. The generator machine kept creating a constant ten gigabit traffic but with a single source address. This allowed us to compare the performance of the selected approaches on a per CPU core basis, as the filter machine's ingress network interface driver selected the same RX queue (and CPU) for all packets.

[3]In reality, such subnets would contain two fewer addresses.

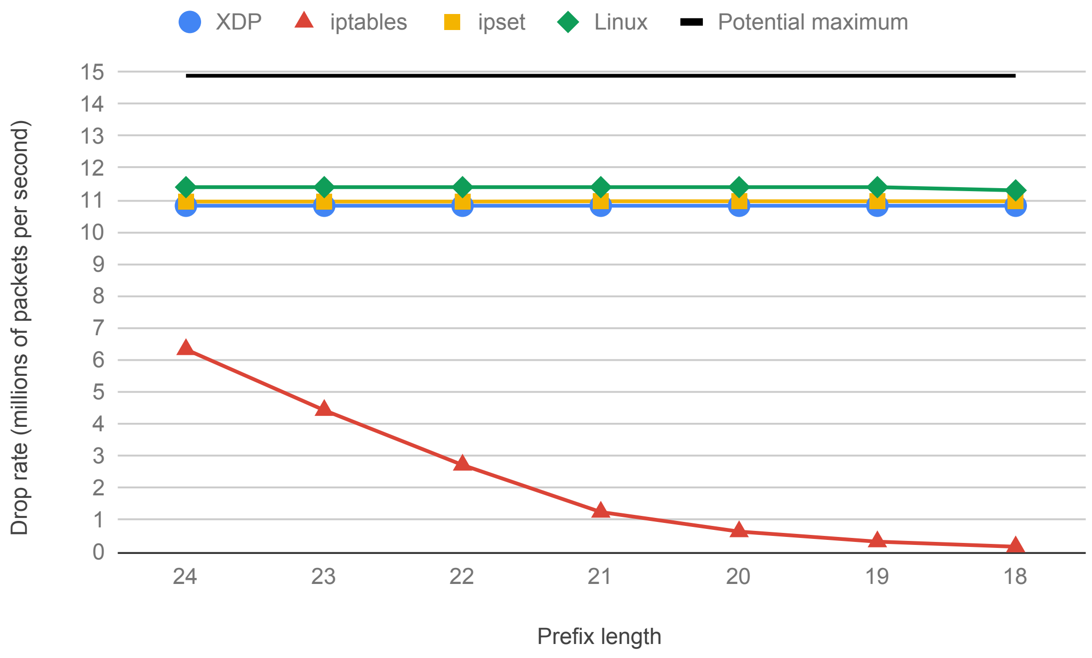

Figure 5.2: Packet drop rate of our XDP filtering solution and three other approaches. Performance is shown based on the size of the source network, i.e., the number of source IP addresses to drop.

Drop	All traffic	One source
XDP	5.4	2.2
iptables	0.88	0.88
ipset	-	0.71
Linux	-	0.63

Table 5.1: Comparison of per CPU core performance of XDP and three other approaches when dropping all incoming traffic and traffic from one source. Values are displayed in million packets per second.

Therefore, the filtering approaches were first configured to drop all incoming packets in the filter machine's ingress interface, and then to drop packets coming one specific source IP address.

The results of the per CPU core testing can be seen in table 5.1. As expected, the XDP solution displayed the best performance with regards to the number of dropped packets in both settings. The `iptables` approach achieve rates of one order of magnitude worse than XDP, while still outperforming `ipset` and the operating system's routing. It also displayed the same results in the two settings, as both settings required it to employ only one filtering rule. In all cases, the one CPU core was running at 100% of its processing capability. The Linux routing table could not be configured to drop all incoming traffic for practical reasons and `ipset` was not capable of specifying a universal IP range.

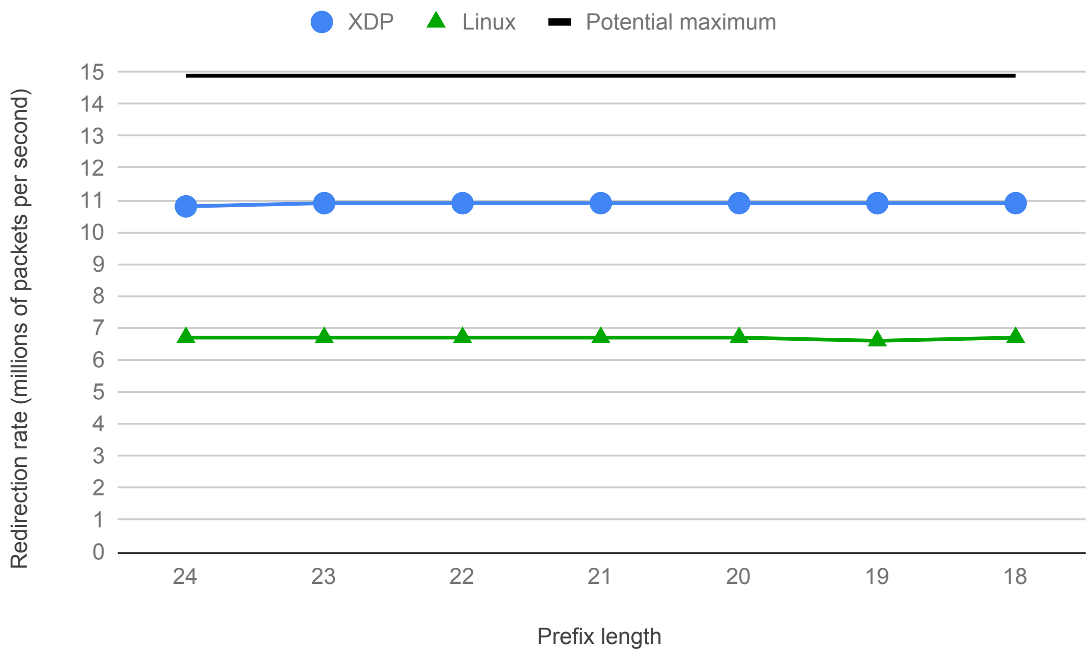

Figure 5.3: Packet redirection rate of our XDP solution and of Linux's routing system. Performance is shown based on the size of the source network, i.e., the number of source IP addresses to redirect.

5.3.2 Packet Redirection

In the following scenario, the filtering approaches were configured to redirect certain traffic from the incoming interface of the filter machine to its egress interface towards the target machine. The traffic ranged, as in the packet dropping scenario in Section 5.3.1, from a /24 sized source network to a /18 sized network.

The approaches' redirection performances were first compared with varying source IP ranges and then in a single source IP setting, comparing them with each other on a per CPU core basis.

Figure 5.3 shows the performance of our XDP filter and of the routing system of the Linux kernel. Both seemed to be capped at a certain rate, XDP at about 11 million packets per second, Linux routing just under 7 million packets per second. Both solutions needed only one entry in their rule set or routing table, where applicable, allowing maximum efficiency. They also did not reach the potential maximum redirection rate, shown as the black bar at 14.88 Mpps, XDP being short of 5 Mpps, Linux routing reaching about half of the potential.

Reaching such relatively low performance in the Linux routing case could have been caused by requiring the threads to access the shared routing table in an exclusive manner which meant that a large proportion of the time the threads were waiting for the routing table to be unlocked by a thread that had been accessing it.

The performance difference between XDP's redirection packet rate and the potential maximum packet rate can also be possibly explained with the shared rule set data structure to which accesses would have needed to be exclusive.

Redirect	All traffic	One Source	Empty map
XDP	2.15	0.62	0.64
Linux	-	0.32	-

Table 5.2: Per CPU core packet redirection in millions of packets per second. Two approaches are shown, the XDP implementation and Linux's routing system. *All traffic* denotes redirecting any incoming packets (no map lookup in the case of XDP), *one source* means redirecting packets coming from one specific host, and *empty map* signifies default XDP redirection of all packets with an empty map lookup.

The CPU load of the XDP program was barely noticeable on the system as it was spread over all 32 cores of the processor. Linux routing's load was spread over the cores equally but because of a high processing overhead, its redirection (or packet forwarding) used about 40% of the CPU's processing capability.

Per CPU Core Performance

The packet redirection scenario was adjusted so that only one CPU core would have been assigned to the tested approaches. The generator machine was set to generate packets from a single source, causing the ingress network interface's driver to choose the same RX queue (and the CPU core) to process all incoming packets.

Results of these tests can be seen in Table 5.2.

First, the XDP program was configured to redirect any incoming packets without regarding the packets' source address. Our solution was able to redirect packets at a rate of 2.15 million packets per second. The Linux routing system could not be configured for such tasks for practical reasons.

Then, the approaches were configured for redirection of packets from a single source address. In the case of XDP, it meant that its rule set map contained one rule specifying to drop packets from the one particular address. In the case of Linux's routing system, a new route based on a source address was added to the interface-specific routing table. The performance results of this configuration were worse in comparison with the first configuration. XDP's performance dropped to about a quarter, 0.62 million packets per second, because of the need to perform a lookup in the LPM trie (albeit with one rule). Linux's routing system displayed a 0.32 million packets per second performance, redirecting packets at about half the rate of XDP.

Finally, XDP was configured to redirect packets from the one source with an empty rule set lookup, redirecting the incoming packets as an implicit rule. At 0.64 million redirected packets per second, its performance increased by about 2% in comparison with the previous configuration. This increase, although not a significant one, shows that a lookup in an empty LPM trie is faster than a lookup in a trie with one rule.

5.3.3 Initialisation Time Consumption

The goal of this scenario was to measure the time complexity of the approaches' initialisations. This scenario was divided into two parts, measuring the time complexity of initialisations to cover certain network sizes, and measuring the time of adding a number of rules to the filtering approaches.

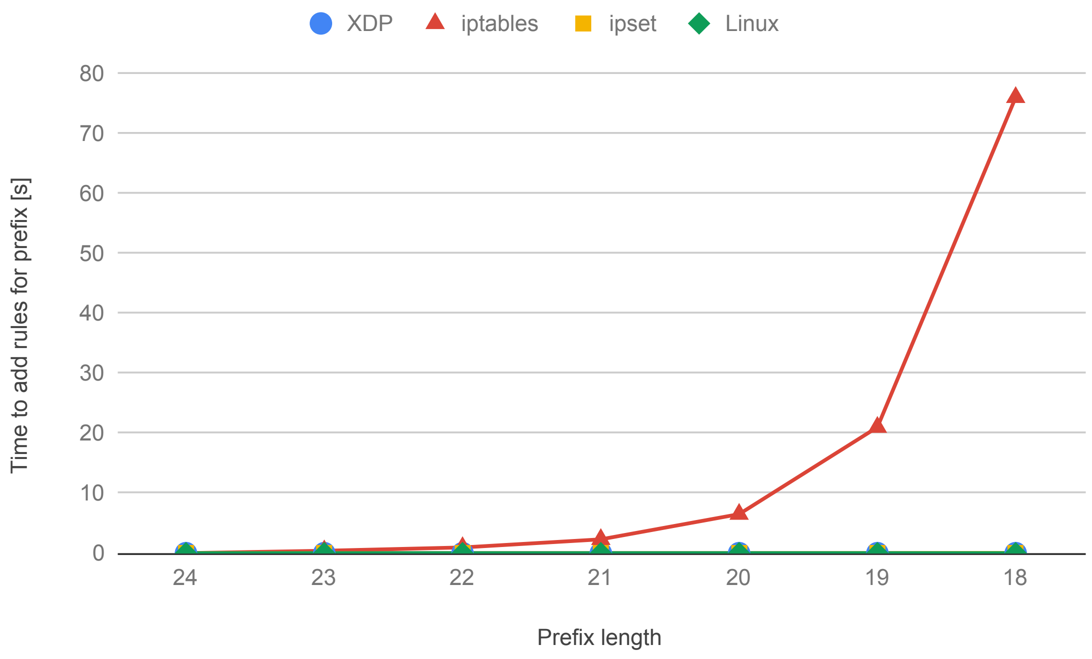

Figure 5.4: Measurements for the amount of time required to initialise the filtering solutions to cover traffic coming from a network of a size expressed with its prefix length.

Time Complexity of Covering Network Ranges

We measured the time required to setup the approaches so that they were ready to filter packets coming from a network of a size ranging from 256 hosts (a /24 network) to 16384 hosts (a /18 network). XDP, `ipset` and Linux's routing table needed only one addition to their filtering data structures (`ipset` also included adding the appropriate IP set to `iptables`), while `iptables` needed to add each source address as a separate rule.

The results of this testing can be seen in Figure 5.4. Because of the nature of their rule addition, XDP, `ipset` and the routing table (denoted as *Linux* in the Figure) performed extraordinarily well, typically requiring no more than 50 milliseconds in any source network size to fill their respective filtering data structures with the appropriate data.

In the case of `iptables`, the time complexity of adding filtering rules grew linearly (shown as exponential in Figure 5.4 because of the exponential x axis). It would not be advisable to employ iptables in time-critical situations where the number of addresses that need to be filtered grows large.

5.3.4 Filtering Entries Addition Time Complexity

In this part, we measured the time required for the filtering approaches to add a number of filtering entries to their appropriate filtering data structures. These entries represented a set of non-overlapping IP networks except in the `ipset` case where one entry represented one source address.

The measurements are displayed in Figure 5.5. All approaches managed to fit their initialisations under one second; sizes of up to 64 entries could be initialised in under 10 milliseconds. The approaches display a linear time complexity of entry additions (the x

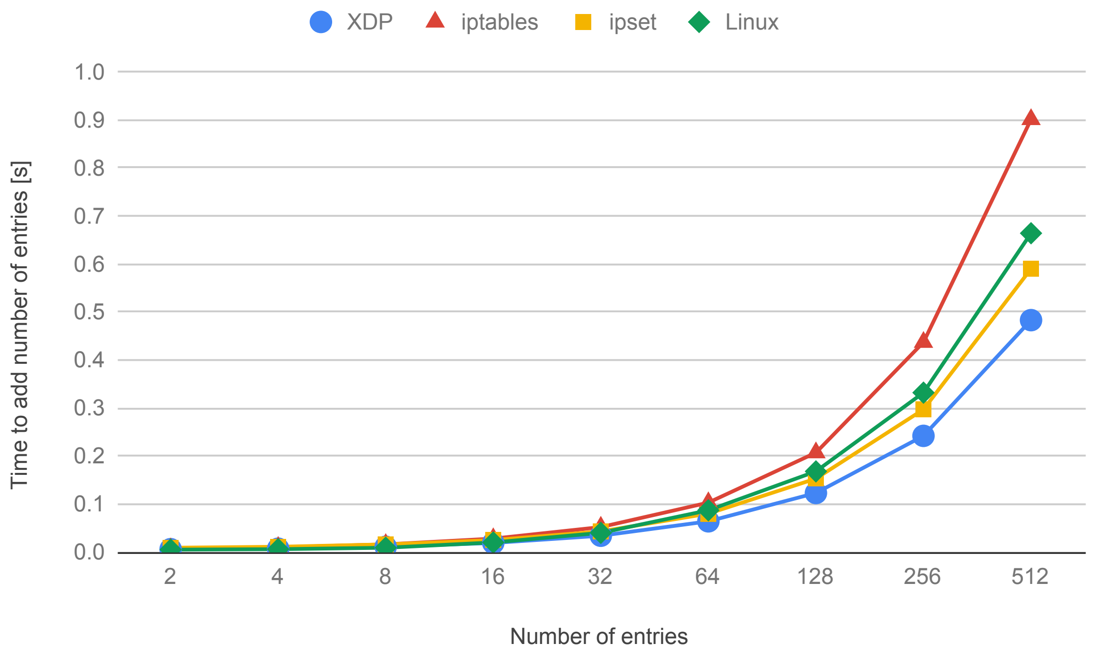

Figure 5.5: Time in seconds required to add a number of entries to the approaches' appropriate filtering data structures.

axis is exponential). Out of the solutions, XDP has proved to be the fastest solution even in the case of rule entry speed. Its addition speed is followed by that of `ipset` and Linux's routing table.

The entry numbers were limited to 512, as that was the maximum number of entries allowed by the eBPF runtime. Measurements of `iptables` additions above this number can be seen in Figure 5.4 from prefix length of 22.

Chapter 6

Conclusion

This work has presented two technologies, namely the (extended) Berkeley Packet Filter (eBPF) and the eXpress Data Path (XDP), which have seen many improvements and increasing support in the latest versions of the Linux kernel. These two technologies can be used in concert to create a high-speed packet filtering system which processes packets as soon as they are received from hardware. The `iptables`, `ipset` and `nftables` tools have been shown as the current widely employed solutions for packet filtering.

A proposal for an eBPF and XDP based packet filtering system has been presented. This system has been implemented and deployed on the NETX router platform developed by Brno University of Technology.

It has been shown that the implementation shows better per-core processing rate than the `iptables`, `ipset` and Linux routing approaches on a number of performance tests. XDP has been able to drop packets at a rate of up to 5.4 million packets per second and to redirect them at a rate of up to 2.15 million packets per second, both being an order of magnitude higher rates than the other tested techniques. It has outperformed the other approaches in all inspected aspects, making it a viable option as a DDoS protection mechanism.

6.1 Future Work

An obvious extension of the filtering solution is the support of processing IPv6 packets. The implemented XDP program, upon detecting that an incoming packet is an IPv6 packet, simply passes the packet to the kernel for further processing. The IPv6 implementation is a straightforward one as the IPv4 approach can be reused with slight alterations to the rule set data type and the packet inspection logic.

Packet processing does not need to be bound only to the network layer. The complete packet data is available to the XDP program for inspection, allowing creation of more complex rules based on certain fields or contents of the transport or the application layers.

Bibliography

[1] bpf(2) - Linux manual page. Last accessed 2019-01-20.
Retrieved from: http://man7.org/linux/man-pages/man2/bpf.2.html

[2] The calculations: 10Gbit/s wirespeed. Last accessed 2019-05-14.
Retrieved from: https://netoptimizer.blogspot.com/2014/05/the-calculations-10gbits-wirespeed.html

[3] Clang: a C language family frontend for LLVM. Last accessed 2019-05-14.
Retrieved from: https://clang.llvm.org/

[4] Intel® Data Direct I/O Technology. Last accessed 2019-05-14.
Retrieved from: https://www.intel.com/content/www/us/en/io/data-direct-i-o-technology.html

[5] Linux Socket Filtering aka Berkeley Packet Filter (BPF). Last accessed 2019-05-14.
Retrieved from: https://www.kernel.org/doc/Documentation/networking/filter.txt

[6] llc - LLVM static compiler. Last accessed 2019-05-14.
Retrieved from: https://llvm.org/docs/CommandGuide/llc.html

[7] Load balancing using iptables with connmark. Last accessed 2019-05-14.
Retrieved from: http://www.system-rescue-cd.org/networking/Load-balancing-using-iptables-with-connmark/

[8] BPF and XDP Reference Guide. 2017. last accessed 2019-05-14.
Retrieved from: http://docs.cilium.io/en/stable/bpf/

[9] Corbet, J.: A JIT for packet filters. Apr 2011. last accessed 2019-01-20.
Retrieved from: https://lwn.net/Articles/437981/

[10] Corbet, J.: BPF: the universal in-kernel virtual machine. may 2014. last accessed 2019-01-20.
Retrieved from: https://lwn.net/Articles/599755/

[11] Dumazet, E.: Re: [PATCH v1] net: filter: Just In Time compiler. Apr 2011. last accessed 2019-01-20.
Retrieved from: https://lwn.net/Articles/437986/

[12] Eychenne, H.: iptables(8) - Linux manual page. Last accessed 2019-01-20.
Retrieved from: http://ipset.netfilter.org/iptables.man.html

[13] Fleming, M.: A thorough introduction to eBPF. Dec 2017. last accessed 2019-01-20. Retrieved from: https://lwn.net/Articles/740157/

[14] Høiland-Jørgensen, T.; Brouer, J. D.; Borkmann, D.; et al.: The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*. CoNEXT '18. New York, NY, USA: ACM. 2018. ISBN 978-1-4503-6080-7. pp. 54–66. doi:10.1145/3281411.3281443.

[15] McCanne, S.; Jacobson, V.: The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX'93. Berkeley, CA, USA: USENIX Association. 1993. pp. 2–2.

[16] Nagy, P.: *Automation of DDoS Attack Mitigation*. Master's Thesis. Brno University of Technology, Faculty of Information Technology. 2018. Retrieved from: http://www.fit.vutbr.cz/study/DP/DP.php?id=20619

[17] NETX: NETX at a glance. Last accessed 2019-01-20. Retrieved from: https://netx.as/netx-at-a-glance/

[18] Russel, P.: The netfilter.org project. 1998. last accessed 2019-01-20. Retrieved from: https://netfilter.org/

[19] Scholz, D.; Raumer, D.; Emmerich, P.; et al.: Performance Implications of Packet Filtering with Linux eBPF. In *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01. Sep. 2018. pp. 209–217. doi:10.1109/ITC30.2018.00039.

[20] Starovoitov, A.: net: filter: rework/optimize internal BPF interpreter's instruction set. Mar 2014. last accessed 2019-01-20. Retrieved from: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8