VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

AKCELERACE APLIKACE PRO POTLAČENÍ DDOS ÚTOKŮ

ACCELERATING AN APPLICATION FOR DDOS MITIGATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — **Bc. KAMIL VOJANEC**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. JAN KUČERA**
SUPERVISOR

BRNO 2022

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačových systémů (UPSY) Akademický rok 2021/2022

Zadání diplomové práce

Student:	**Vojanec Kamil, Bc.**
Program:	Informační technologie a umělá inteligence
Specializace:	Vestavěné systémy
Název:	**Akcelerace aplikace pro potlačení DDoS útoků** **Accelerating an Application for DDoS Mitigation**
Kategorie:	Počítačová architektura

Zadání:

1. Seznamte se s knihovnou DPDK, klasifikačním rozhraním rte_flow a aplikací vyvíjenou v rámci sdružení CESNET pro ochranu před DDoS útoky.
2. Analyzujte možnosti frameworku DPDK s ohledem na jeho využití k akceleraci této aplikace. Zaměřte se na možnosti přesunu klasifikačních pravidel na síťovou kartu.
3. Navrhněte vhodný způsob akcelerace aplikace pro vybrané síťové karty podporující DPDK.
4. Proveďte implementaci navrženého řešení a jeho funkčnost ověřte na dostupné platformě.
5. Diskutujte dosažené výsledky a možnosti dalšího pokračování práce.

Literatura:

- Podle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce:	**Kučera Jan, Ing.**
Konzultant:	Viktorin Jan, Ing., CESNET
Vedoucí ústavu:	Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání:	1. listopadu 2021
Datum odevzdání:	18. května 2022
Datum schválení:	29. října 2021

Abstrakt

Diplomová práce se zabývá optimalizací a akcelerací aplikace pro potlačení útoků typu odepření služby. Cílem práce je analyzovat existující implementaci aplikace DDoS Protector a identifikovat součásti, které je vhodné optimalizovat nebo akcelerovat. Na základě této analýzy je proveden návrh nového přístupu ke klasifikaci paketů s využitím open-source frameworku *DPDK* a návrh hardwarové akcelerace pomocí knihovny *RTE Flow*. Výsledkem této práce je sada modulů a implementace nezbytných komponent pro aplikaci DDoS Protector. Výsledné komponenty jsou pak řádně testovány. Na závěr je provedeno srovnání výsledků původní a nové implementace. Například při použití 256 mitigačních pravidel dochází s upravenými komponentami až k pětinásobnému zvýšení paketové propustnosti celé aplikace.

Abstract

This thesis focuses on optimizing and accelerating an application used for mitigating Denial of Service attacks. The goal is to analyze the existing implementation of DDoS Protector and to identify components which are suitable for optimization or hardware acceleration. Based on the analysis, improved algorithms and data structures utilizing the *DPDK* open-source framework are designed together with a proposal to offload certain computation elements into hardware using the *RTE Flow* library. The result of this thesis is a set of modules and an implementation of classification components intended to be used within the DDoS Protector application. The resulting components are then properly tested. Finally, the performance results of the original and new implementations are compared. The application shows as much as five-times improvement in terms of packet rate when using 256 classification rules.

Klíčová slova

DDoS, Odepření služby, DPDK, RTE Flow, Hardwarová akcelerace

Keywords

DDoS, Denial of Service, DPDK, RTE Flow, Hardware acceleration

Citace

VOJANEC, Kamil. *Akcelerace aplikace pro potlačení DDoS útoků*. Brno, 2022. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jan Kučera

Akcelerace aplikace pro potlačení DDoS útoků

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Jana Kučery. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Kamil Vojanec
13. května 2022

Poděkování

Chtěl bych poděkovat svému vedoucímu Ing. Janu Kučerovi za vedení a odborné konzultace při zpracování této práce. Dále bych chtěl poděkovat Ing. Janu Viktorinovi a ostatním členům oddělení Nástrojů pro administraci a bezpečnost při sdružení CESNET. Zároveň bych chtěl poděkovat své rodině za jejich podporu po celou dobu mého studia.

Obsah

Kapitola 1

Úvod

S rostoucím objemem provozu v počítačových sítích dochází také ke zvýšení rychlosti přenosu, což vede na potřebu stále výkonnějších zařízení, které dokáží provoz efektivně zpracovávat. Jedním z nezbytných důvodů pro potřeby vysokorychlostního zpracování provozu je i zabezpečení počítačových sítí proti různým druhům útoků. Mezi ně patří i útoky typu odepření služby (*Denial of Service, DoS*), případně jejich distribuovaná varianta (*Distributed Denial of Service, DDoS*).

Pro zajištění efektivního zpracování paketů ve vysokorychlostních sítích je pak možné síťové aplikace dále akcelerovat přesunem části zpracování přímo do hardware síťové karty (tzv. *offload*).

Přímé využívání hardware síťové karty je však s jejich existujícími ovladači velmi obtížné a funguje pouze pro konkrétní síťové karty. Tyto problémy částečně řeší sada knihoven *DPDK*, která implementuje vrstvu abstrakce jak pro zpracování paketů v software, tak pro ovládání některých hardwarových funkcí síťových karet.

Tato práce se zabývá optimalizací a akcelerací aplikace pro ochranu před DDoS útoky. Jejím cílem je pomocí profilovacích technik analyzovat existující implementaci aplikace DDoS Protector a identifikovat komponenty, které je vhodné optimalizovat pomocí specializovaných knihoven frameworku DPDK. Dalším cílem je pak implementovat nové komponenty, které povedou ke zvýšení efektivity aplikace a toto zvýšení výkonu ověřit.

Výsledkem práce je návrh několika variant klíčových komponent aplikace DDoS Protector a implementace navržených modulů. Takto implementované moduly mohou být následně přímo začleněny v rámci aplikace.

Popsané přístupy k optimalizaci a akceleraci jsou však univerzální a vytvořené komponenty je možné využít při implementaci jiných aplikací. Posledním výsledkem této práce je pak analýza dosažených výsledků ve spojení s testováním funkčnosti a výkonnosti optimalizované aplikace.

Práce je členěna následovně. Útoky typu odepření služby, jejich popisem a klasifikací se zabývá úvodní kapitola 2. Použitým frameworkem *DPDK*, jeho klíčovými vlastnostmi, součástmi a možnostmi hardwarové akcelerace se pak věnuje kapitola 3. Následující kapitola 4 se věnuje analýze vlastností existující aplikace pro ochranu před DDoS útoky a identifikaci komponent vhodných k optimalizaci. Návrhu samotné optimalizace a akcelerace vybraných komponent pomocí *RTE Flow* se pak zabývá kapitola 5. Popisu navržených komponent, rozboru použitých algoritmů a datových struktur v souvislosti s jejich implementací se dále věnuje kapitola 6. Kapitola 7 se dále zaměřuje na testování implementovaných modulů, měření výkonnosti optimalizovaných komponent a analýzu získaných výsledků. Závěrečná kapitola 8 nakonec zhodnocuje výsledky této práce.

Kapitola 2

Útoky typu odepření služby

Útoky typu odepření služby (*Denial of Service*) se snaží cíleně narušit běh služeb či sítí pomocí zahlcení síťových linek nebo výpočetních zdrojů serverů velkým množstvím paketů. Takové narušení má za důsledek, že veškeré legitimní požadavky klientů k dané službě jsou zahozeny [18].

Zahlcení síťových zdrojů z jednoho počítače je však v dnešní době téměř nemožné, a tak se častěji objevují distribuované varianty útoků (*Distributed Denial of Service*, *DDoS*). Ty rozšiřují myšlenku DoS útoků o spojení několika útočících strojů, které pak zahlcení provádějí zároveň. Takto paralelní útok je již schopen mít za cíl jakékoliv síťové služby.

DDoS útok je typicky rozfázován do 4 kroků:

1. Útočník vybírá počítače (tzv. agenty nebo zombie), které budou následně provádět samotný útok.

2. Vytipovaní agenti jsou napadeni škodlivým kódem, který přichystá agenta k provedení útoku a ustaví spojení s útočníkem pomocí pomocných počítačů (tzv. *handlerů*). Tímto vzniká síť útočících počítačů, rovněž nazývaná jako *botnet*.

3. Agenti potvrdí připravenost k útoku svým handlerům a ti pak samotnému útočníkovi.

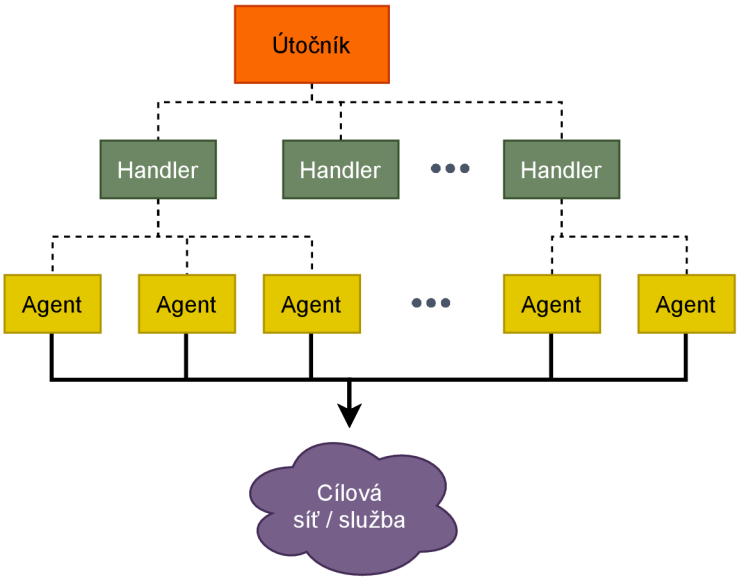
4. Útočník vydá příkaz k útoku a jednotliví agenti individuálně útok vykonávají.

Obrázek 2.1 znázorňuje hierarchickou organizaci DDoS útoků. Tučnou plnou čárou je znázorněn samotný útok pocházející od jednotlivých agentů. Přerušovanou čarou je pak naznačena komunikace předcházející samotnému útoku.

Jednotlivých agentů v rámci botnetu mohou být až desítky tisíc. Například síť *Mēris*, objevená v červnu 2021 zahrnuje alespoň 56000 agentů, avšak celkový odhadovaný počet napadených zařízení dosahuje až 200000. Napadenými agenty botnetu *Mēris* byly routery společnosti Mikrotik [11].

Vystopovat původce DDoS útoků je z důvodu jejich distribuované povahy téměř nemožné. V současné době se tak přistupuje k potlačení (*mitigaci*) dopadů těchto útoků pomocí monitorování provozu v síti a zahazování paketů, které pravděpodobně patří k útoku. Tento způsob je však ve vysokorychlostních sítích výpočetně náročný a vyžaduje specializované aplikace optimalizované pro tento typ úlohy.

Distribuované útoky odepření služby mohou být prováděny různými způsoby. DDoS útoky je možné rozdělit na útoky volumetrické, útoky využívající specifických vlastností protokolu a útoky na vlastnosti aplikací. Informace o jednotlivých typech útoků jsou převzaty zejména z [22], [28] a [18].



Obrázek 2.1: Schéma typické organizace DDoS útoku.

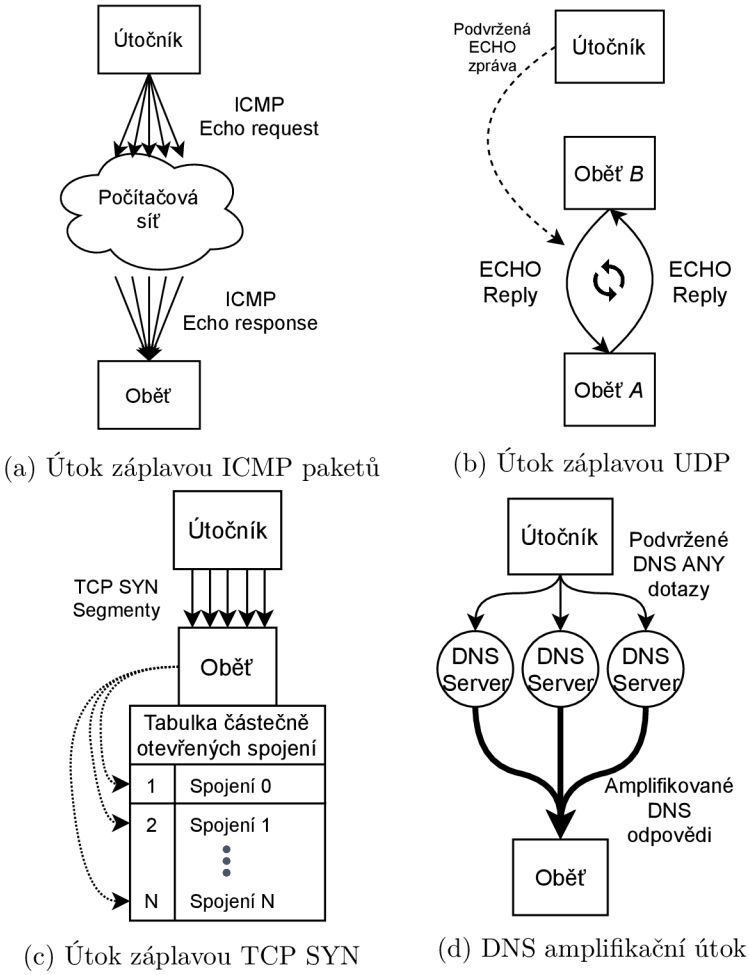
2.1 Volumetrické útoky

Volumetrické útoky jsou nejjednodušším typem DDoS útoků. Jejich principem je generování obrovského objemu síťového provozu, který saturuje síťovou linku, čímž je odstavena celá připojená síť. Realizace těchto útoků může probíhat několika způsoby:

Záplava ICMP paketů (ICMP Flood) Základní princip tohoto útoku je graficky znázorněn na obrázku 2.2a. Vidíme, že útočník zasílá velké množství ICMP zpráv typu *echo request* se zdrojovou IP adresou podvrženou tak, že je nastavena na adresu počítače, který je cílem útoku. Jednotlivé zařízení, které takto podvrženou zprávu obdrží pak generují ICMP zprávy *echo reply*, které ale zasílají na adresu oběti útoku. Pokud je takových zpráv vygenerován dostatečný počet, je oběť zaplavena ICMP zprávami natolik, že přestává správně fungovat. Tento útok by se rovněž mohl řadit mezi útoky využívající vlastnosti protokolů.

Záplava UDP paketů (UDP Storm) Princip UDP Storm útoku je zobrazen ve schématu na obrázku 2.2b. Vidíme, že se útok podobá výše zmíněnému ICMP útoku. Hlavním rozdílem je v tomto případě využití UDP *echo portu* dvou odlišných zařízení. Útočník nejprve podvrhne zdrojovou adresu jedné ze dvou obětí útoku (nazvěme ji A) paket s touto adresou zašle na echo port druhé z obětí (B). Ta na tento paket reaguje vrácením paketu se stejnými daty opět stroji A. Cyklus dotazů a odpovědí mezi oběma stroji se pak může opakovat do nekonečna a může omezovat legitimní provoz.

Amplifikační útok (Reflector Attacks) Amplifikační útoky se mohou považovat za jistou variantu jak volumetrických útoků, tak i útoků s využitím vlastností protokolů. Principem amplifikačních útoků je využití určitých síťových služeb (např. DNS) k zesílení (*amplifikaci*) mohutnosti samotného útoku. Ilustrace tohoto typu útoku s využitím služby DNS je ilustrována na obrázku 2.2d. Vidíme, že útočník podvrhne dotaz na zvolenou službu, avšak odpověď na zaslanou zprávu bude doručena na IP adresu



(a) Útok záplavou ICMP paketů

(b) Útok záplavou UDP

(c) Útok záplavou TCP SYN

(d) DNS amplifikační útok

Obrázek 2.2: Znázornění principu některých typů DDoS útoků.

oběti útoku. Pokud je navíc odpověď na původní dotaz objemově větší, než původní zpráva, pak dochází ke zmíněné amplifikaci. V souvislosti s tímto typem útoků je definován koeficient *BAF* (*Bandwidth Amplification Factor*), který značí poměr velikosti odpovědi, která je doručena oběti, a velikosti původního dotazu [13].

Jedním z protokolů, který je často používán k realizaci amplifikačních útoků je DNS (*Domain Name System*). Ten poskytuje dotaz *ANY* žádající doménový server o veškeré informace spojené se zvolenou doménou [1]. Odpovědi na takové dotazy mohou být dle [13] 28× až 54× větší než původní dotaz (BAF se tedy pohybuje mezi 28 a 54).

2.2 Útoky na vlastnosti protokolů

Útoky tohoto typu také často využívají záplavy některých typů paketů. Oproti klasickým útokům na bázi objemu však cílí na konkrétní vlastnosti zvolených protokolů k usnadnění zahlcení cílového počítače. Na rozdíl od volumetrických útoku sice nemusí být zahlcena samotná síťová linka, dochází však k zaplnění paměti nebo přetížení výpočetních jednotek cílového stroje, což zabraňuje zpracování legitimního provozu. Typickými útoky tohoto typu jsou například:

Záplava TCP SYN segmentů (SYN Flood) SYN Flood útok je ilustrován na obrázku 2.2c. Tento útok využívá vlastnosti TCP protokolu, konkrétně mechanismu navázání spojení. Útočník zaplaví oběť TCP segmenty s nastaveným příznakem SYN, což na straně oběti způsobí alokaci položek v tabulce částečně otevřených spojení. Tato tabulka je však omezená a při jejím přeplnění začne docházet k postupnému odstraňování nejstarších záznamů. Pokud tedy útočník dokáže SYN segmenty zasílat častěji, než oběť dokáže záznamy odstraňovat, dojde k úplnému zahlcení tabulky a legitimní provoz nebude obsloužen.

Podle [23] tvořily SYN Flood útoky ve třetím čtvrtletí roku 2021 více než polovinu všech zaznamenaných DDoS útoků.

Ping-of-death Útoky Ping-of-death používají pro zahlcení stroje oběti pouze jeden paket. Jedná se o paket typu ICMP *echo request*, který je vytvořen tak, že po přijetí zabírá více než je maximální specifikovaná velikost paketu na síťové vrstvě. Takto velký paket pak může způsobit přetečení v datových strukturách použitých operačním systémem oběti a zpomalení či zablokování příjmu dalších paketů. Tento útok v dnešní době již téměř vymizel, protože moderní operační systémy jsou vybaveny lepšími ochrannými mechanismy.

2.3 Útoky na aplikace

Jedná se o nejsložitější typ DDoS útoků, který od útočníka vyžaduje znalost konkrétní služby nebo aplikace, na kterou cílí. Útoky tohoto typu často nevyžadují příliš vysokou rychlost provozu a proto mohou být také nazývány jako pomalé útoky. Tyto útoky nejčastěji cílí na webové servery, jako například *Apache*. Mezi útoky DDoS útoky na aplikace patří:

Slow Loris Generuje malý objem provozu s protokolem HTTP, který po částech posílá na cílový webový server. HTTP zpráva však není nikdy ukončena a server tak musí držet spojení nekonečně dlouhou dobu. Takto nekonečných spojení může být vytvořeno několik, což vede k pomalému vyčerpání paměti a výpočetních prostředků oběti.

R.U.D.Y. Útoky R.U.D.Y. (*R-U-Dead-Yet*) jsou podobně jako Slow Loris útoky s nízkou přenosovou rychlostí využívající HTTP protokolu. Narozdíl od Slow Loris ale R.U.D.Y. zasílá na server oběti legitimní HTTP zprávy typu *POST*. Ty jsou ale rozděleny do velkého množství paketů s dlouhou hlavičkou a velmi krátkým tělem. Takto rozdělené zprávy nutí webový server udržovat spojení po velmi dlouhou dobu a tím blokovat legitimní provoz.

Kapitola 3

Framework *DPDK*

Data Plane Development Kit, známější pod zkratkou *DPDK*, je open-source platforma (*framework*) obsahující sadu knihoven určených k tvorbě aplikací pro zpracování síťových paketů. Významnou vlastností DPDK je možnost provádět zpracování síťového provozu v uživatelském prostoru (*user-space*) operačního systému a plně obejít jádro (*kernel*), což je znázorněno na obrázku 3.1. Díky tomu je možné zamezit časově náročnému přepínání kontextu mezi aplikací a jádrem, což má za důsledek zvýšení výkonu.

Na schématu je také vidět, že DPDK namísto síťových ovladačů poskytovaných operačním systémem využívá tzv. *poll mode driver* (viz sekce 3.1). Dalšími vlastnostmi vhodnými pro rychlé zpracování síťového provozu je efektivní správa paměti pomocí velkých stránek (*huge pages*) (sekce 3.2), rozložení provozu mezi více procesorových jader a vektorizace (sekce 3.3).

Informace uvedené v této kapitole vycházejí zejména z publikací [17], [15], [29].

3.1 Síťové ovladače

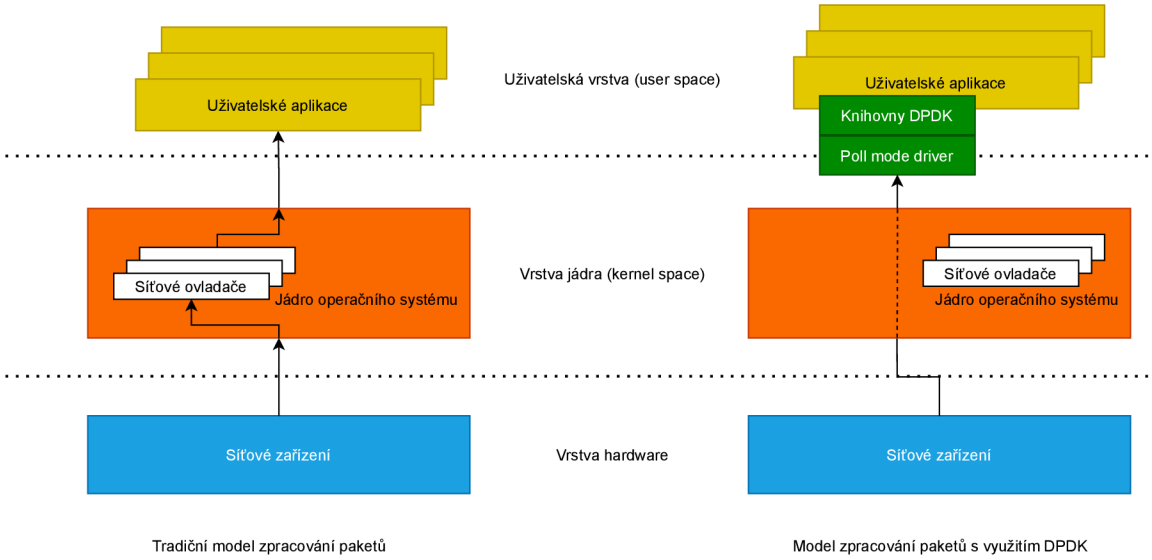
Ovladače zařízení (*device drivers*) v operačních systémech jsou speciální softwarové moduly, které poskytují vrstvu abstrakce nad konkrétním zařízením. Z pohledu uživatele těchto zařízení je pak možné na ovladač nahlížet jako na černou skříňku, která poskytuje specifikované rozhraní pro komunikaci se zařízením a kterou je možné připojovat a odpojovat za běhu systému.

V případě síťových zařízení můžeme rozlišit dva principy implementace ovladačů:

Interrupt (*přerušení*) Přerušení je speciální signál, který může zařízení vyslat procesoru počítače. Síťové karty přerušení vyvolávají například v případě, že do jejich vstupní fronty (*RX queue*) vstoupí jeden nebo více paketů. V rámci obsluhy přerušení pak ovladač síťové karty zprostředkuje přenos dat ze síťové karty do speciálních struktur v operační paměti počítače pomocí DMA.

Tento způsob zpracování paketů byl historicky dostačující, jelikož rychlost síťových přenosů byla nižší, než dnes. Při vysokorychlostních přenosech však může docházet k zaplnění vstupní fronty, ztrátě paketů a v nejhorším případě i k zablokování systému z důvodu opakovaného vyvolávání přerušení ze síťové karty.

Polling Polling je technika, při které se procesor opakovaně dotazuje na přítomnost nových dat. V případě síťových zařízení se dotazuje prostřednictvím ovladače na data ve vstupní frontě. Z tohoto důvodu není polling příliš vhodný pro použití v sítích



Obrázek 3.1: Schéma porovnávající tradiční způsob zpracování paketů v operačním systému se způsobem, který využívá DPDK.

s nižší frekvencí příchozích paketů. Naopak ve vysokorychlostních sítích jsou data ve frontě síťové karty k dispozici téměř nepřetržitě a pollování tak umožňuje přijímat pakety nejvyšší možnou rychlostí, kterou procesor dovoluje, aniž by byl přitom zatěžován obsluhou přerušení.

Jádro operačního systému Linux tradičně využívalo ovladače s mechanismem přerušení, avšak v novějších verzích je již poskytnuto rozhraní *NAPI* (*New API*) [12], které umožňuje ve vybraných případech a u podporovaných zařízení vypnout zpracování přerušení a síťová data přijímat jen pomocí mechanismu polling. Oba zmíněné přístupy v systému GNU/-Linux přistupují k ovladačům jako k modulům samotného jádra a pro zpracování paketů aplikacemi je nutné přepínat kontext mezi jádrem a aplikací.

Naproti tomu DPDK implementuje vlastní ovladače PMD (*Poll Mode Driver*). Tyto ovladače jsou implementovány pro každé podporované síťové zařízení a jsou umístěny do uživatelského prostoru (*user-space*), což zamezuje nutnosti přepínat kontext mezi aplikací a jádrem systému. PMD zpravidla využívají principu polling, avšak je možné je přepnout do režimu přerušení.

Poll mode drivery v DPDK navíc zpracovávají pakety ve skupinách (*burst*) pro další optimalizace výkonu aplikace.

3.2 Správa paměti

Každý běžící proces získává od operačního systému přidělen určitý úsek tzv. *virtuální paměti*. Virtuální paměť je abstrakce umožňující navenek oddělit adresní prostor, který je využíván aplikacemi od skutečného rozložení fyzické paměti. Mezi adresami ve fyzické a virtuální paměti je však nutné překládat. K tomuto účelu je v procesorech speciální hardwarová jednotka MMU (*Memory Management Unit*), která je využívána mechanismem tzv. *stránkování* (*paging*). Každému procesu tak mohou být přiděleny individuální stránky paměti. Běžná velikost přidělovaných stránek paměti je 4 kB.

Pro aplikace s využitím DPDK je však vhodné alokovat daleko větší stránky pro uložení dat. DPDK tak využívá mechanismu tzv. *velkých stránek* (*huge pages*). Tyto stránky mohou mít velikost 2 MB nebo 1 GB. Hlavní výhodou použití velkých stránek pro alokaci paměti je snížení počtu výpadků stránek v jednotce *Translation Look-aside Buffer* (*TLB*), což má za důsledek snížení latence přístupu do paměti a zvýšení výkonu. Implementace velkých stránek je provedena ve speciálním souborovém systému – *hugetlbfs*, ve kterém každý soubor reprezentuje jednu velkou stránku. Alokace velké stránky je pak provedena načtením příslušného souboru do paměti [20].

Při spuštění aplikace s využitím DPDK jsou tak nejprve alokovány potřebné velké stránky a z těchto stránek jsou pak samotné aplikaci přidělovány bloky paměti pomocí specializovaného alokátoru. V uživatelském programu je pak namísto volání standardní funkce *malloc* jazyka C volána speciální funkce *rte_malloc*. Mezi výhody použití specializovaného alokačního algoritmu DPDK patří [16]:

- Možnost alokovat velké stránky.
- Alokace bere v úvahu nerovnoměrnou přístupovou dobu různých jader procesoru k různým paměťovým adresám (*Non-Uniform Memory Architecture, NUMA*).
- Paměť je přidělována zarovnaná na velikost položky vyrovnávací paměti procesoru (*cache line alignment*).
- Přidělená paměť je sdílená napříč všemi jádry procesorů.
- Procesy mohou žádat o alokaci souběžně.
- Alokace, přístup a dealokace jednoho bloku přidělené paměti může být provedena v různých procesech.

V případě alokace často používaných struktur pevné velikosti je navíc v DPDK obvykle používán mechanismus tzv. *memory pool*. Memory pool je objekt, který předalokuje určitý počet struktur pevné velikosti a ty potom poskytuje dalším objektům. Namísto dealokace je pak poskytovaná struktura předána zpět objektu memory pool, který jej tak může vydat dalším žadatelům nebo bezpečně dealokovat. Tento způsob je nejčastěji využíván při alokaci struktur pro pakety (*rte_mbuf*) [8].

3.3 Paralelismus a vektorizace

Platforma DPDK poskytuje pro spouštění aplikací s více vlákny modul *EAL* (*Environment Abstraction Layer*). Cílem této vrstvy je poskytnout abstrakci a odstínit kód aplikace od detailů platformy, na níž aplikace běží. V prostředí DPDK jsou tak vlákna interně implementována pomocí knihovny *pthread* pocházející ze standardu *POSIX*. Tato vlákna se však v DPDK nazývají jako *lcores* a jejich spouštění má na starosti právě EAL.

Pro zvýšení výkonu aplikací provádí EAL navíc tzv. *core pinning*, což vynutí spuštění daného vlákna na konkrétním procesorovém jádře. Stírá se tak rozdíl mezi softwarovým vláknem a hardwarovým jádrem procesoru. Zároveň se eliminuje režie spojená s migrací vláken mezi jádry a přepínáním kontextu.

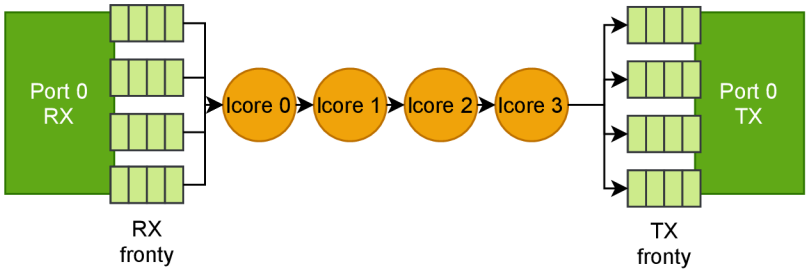
DPDK využivá *master – slaves* modelu vláken. Vlákno, v němž je spuštěn proces DPDK aplikace se stává master vláknem. Inicializace pak probíhá zavoláním inicializační rutiny EAL, která je pak zodpovědná za zjištění vlastností procesoru (zejména počtu dostupných

jader), spuštění vláken typu slave a případný core pinning těchto vláken na jádra. Rozdělení práce mezi jádra a komunikace mezi nimi je stále v režii návrháře aplikace.

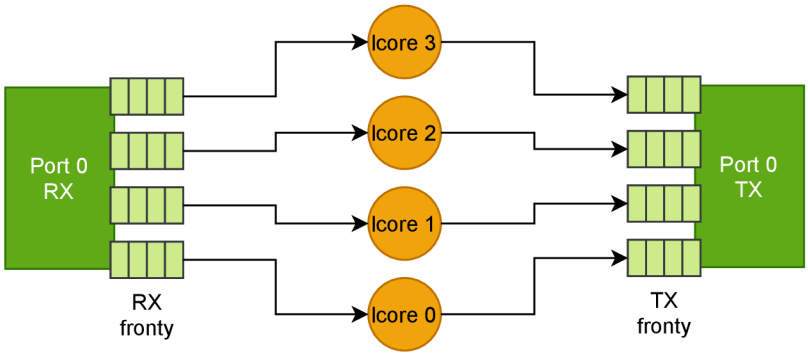
DPDK podporuje dva základní způsoby rozdělení práce mezi lcores [29]:

- *Pipelined model* (obrázek 3.2a) využívá přístupu zřetězené linky. Pakety ze všech front jsou tedy předány jednomu jádru, které je částečně zpracuje a předá k dalšímu zpracování následujícímu jádru. Takto pakety projdou postupně všemi jádry a jsou posledním jádrem vloženy do zvolené výstupní fronty.
- *Run-to-completion model* (obrázek 3.2b) rozděluje pakety z vstupních front mezi jednotlivá procesorová jádra, každé jádro pak nezávisle přidělené pakety zpracuje a vloží do přidělené výstupní fronty.

Pro komunikaci mezi vlákny poskytuje DPDK několik knihoven, například *rte_ring*, která implementuje kruhový buffer (*ring*), k němuž je možné přistupovat zcela bez potřeby zamykání [7].



(a) Pipelined model.



(b) Run-to-completion model.

Obrázek 3.2: Modely zpracování síťového provozu na úrovni lcores v DPDK.

Kromě paralelismu na úrovni jader se v knihovnách DPDK používá i datový paralelismus s pomocí vektorových jednotek moderních procesorů *SIMD*. Specializované komponenty (sekce 3.5) pak zpravidla implementují několik variant pro různé generace vektorových jednotek od starších *SSE* až po nejmodernější *AVX-512*. Při použití těchto komponent je pak možné na úrovni programu tyto varianty přepínat.

3.4 Knihovna *RTE Flow*

Pro akceleraci zpracování paketů ve vysokorychlostním provozu často nedostačuje výkon procesoru a je tak vhodné provést převod části zpracování paketů do specializovaného hard-

ware na síťové kartě. Tento postup se nazývá *offload*. Moderní síťové karty dnes poskytují několik možností offloadu výpočtu [29]:

- *Receive Side Scaling* (*RSS*) je technika rozdělení příchozích paketů mezi různé vstupní fronty. K tomuto účelu musí být paket nejprve rozdělen na jednotlivé hlavičky a vybrané hlavičky jsou poté hashovány vhodnou hashovací funkcí. Typicky se pro tento účel používá funkce *Toeplitz hash* [4]. Takto vypočítaná hodnota hashe pak určuje index fronty, do níž je příslušný paket vložen.

  Speciálním případem RSS je *Flow Director* (*FDIR*), který umožňuje specifikovat síťové toky, jejichž pakety budou vloženy do konkrétních front pro separaci těchto toků. Použití FDIR je vhodné například pro přiřazení front konkrétním aplikacím.

- *VLAN Insert* a *VLAN Strip* umožňují vkládat a odebrat hlavičky VLAN nebo podle jejich hodnot filtrovat příchozí pakety. Vložení nebo odstranění hlavičky VLAN způsobuje změnu struktury paketu a je nutné přepočítat kontrolní součet L2.

- *Timestamping* (časové značkování) přiřazuje každému paketu časovou značku. Pro spuštění značkování paketů je nejprve provedena synchronizace s časovým serverem pomocí protokolu *Precision Time Protocol* (IEEE1588).

- Výpočet a kontrola správnosti kontrolních součtů na vrstvách L2 (*Ethernet CRC*), L3 (*IP checksum*), L4 (*TCP, UDP, SCTP checksum*). V případě detekce chybného kontrolního součtu je tento jev signalizován nastavením speciálního bitu v DPDK reprezentaci paketu (*rte_mbuf*).

- Obecná klasifikace toků podporuje zpracování jednotlivých hlaviček paketů a aplikaci definovaných akcí pro klasifikované pakety. Tato funkcionalita umožňuje provádět libovolnou manipulaci s hlavičkami paketů nebo zahazování paketů.

DPDK poskytuje několik způsobů konfigurace offloadů. Pro většinu zmíněných offloadů je poskytováno specializované rozhraní jejich konfigurace. Navíc však existuje generické rozhraní v podobě knihovny RTE Flow.

Pravidla RTE Flow

Offloadované operace v hardware mají formu pravidel. Příklad struktury pravidla je zobrazen v tabulce 3.1. Vidíme, že pravidla se skládají ze 3 částí – *atributy*, *matching pattern* a *akce*.

Atributy

Specifikují vlastnosti daného pravidla. Struktura atributů se dále skládá ze 4 prvků:

Group Určuje číslo skupiny, do které pravidlo náleží (pokud daná síťová karta podporuje rozdělení do více skupin). Skupiny umožňují rozdělit pravidla do hierarchie tabulek, které se mohou zpracovávat postupně.

Priority Číslo, kde hodnota 0 značí nejvyšší prioritu. Pokud existují dvě pravidla, která mají stejný *matching pattern*, je pak aplikováno to pravidlo, které má nižší hodnotu priority. Podobně jako u atributu group může být rozsah úrovní priorit omezen konkrétní síťovou kartou.

Atributy	Název	Hodnota			
	Group	1			
	Priority	1			
	Traffic direction	ingress			
	Transfer	0			
Matching Pattern	Index	Pattern Item	Spec	Mask	Last
	0	Ethernet	04:3f:72:c7:b8:84	ff:ff:ff:ff:ff:ff	0
	1	IPv4	10.113.172.69	255.255.255.0	0
	2	TCP	80	0	89
	3	END	0	0	0
Akce	Index	Název	Argumenty		
	1	DROP	0		
	2	END	0		

Tabulka 3.1: Příklad definice RTE Flow pravidla, které zahazuje klasifikované pakety.

Traffic direction Specifikuje směr provozu, na nějž je pravidlo aplikováno. Pravidlo tak může být aplikováno na pakety příchozí (*ingress*) nebo odchozí (*egress*).

Transfer Pokud je tento atribut nastaven, pravidlu je umožněno přesměrovat provoz mimo samotnou aplikaci přímo na úrovni síťové karty.

Matching pattern

Definuje vzory v jednotlivých hlavičkách paketu, pomocí nichž bude paket klasifikován. Celkově se skládá z jednotlivých *pattern items*, které jsou pak sestaveny do celkového vzoru. Sestavení je závislé na pořadí a pro správnou funkčnost je třeba umístit na první pozice vzory pro nejnižší síťové vrstvy. Jednotlivé prvky vzoru (pattern items) tedy mohou specifikovat buď samotné hlavičky paketu nebo určitá metadata, která nejsou součástí samotného paketu. Formát každého prvku vzoru je určen až třemi strukturami stejného typu:

1. *spec* značí hodnoty, které se v paketech mají vyskytovat.
2. *last* určuje horní hranici pro porovnání při klasifikaci pomocí rozsahů.
3. *mask* specifikuje bitovou masku, která bude aplikována na hodnoty *spec* a *last*.

Pro správné použití pattern item pro hlavičky paketů je třeba specifikovat alespoň hodnotu *spec* a jednu ze dvou ostatních. Některé typy *pattern item* však nemusejí podporovat zadávání hodnoty *last*, zatímco použití *mask* je zpravidla funkční všude.

V tabulce 3.1 je matching pattern složen ze 4 prvků. V případě položek Ethernet a IPv4 vidíme použití pouze *mask*, u TCP je pak zobrazeno použití *last*. Rovněž na indexu 3 je vidět prvek *END*, který je speciální a je považován za zarážku, která zakončuje daný matching pattern. Tento prvek musí být obsažen v každém patternu a nepotřebuje *spec*, *mask* ani *last*. V tabulce je také vidět seřazení prvků podle síťové vrstvy.

Akce

Definuje operace, které má dané pravidlo provést. Každé pravidlo může obsahovat několik akcí, které jsou provedeny v pořadí, ve kterém jsou zapsány. Při definici akcí je třeba vždy

specifikovat tzv. *fate* (osud) paketu, který je zpracováván daným pravidlem. Ten určuje, co se s paketem stane na výstupu zpracování (například zahození nebo zařazení do výstupní fronty). Podobně jako u definice matching patternu musí i definice akcí končit speciální akcí *END*. Mezi často definované akce patří:

DROP Tato akce nevyžaduje žádné argumenty a slouží k zahození zpracovaného paketu. Touto akcí je možné specifikovat osud paketu.

RSS Vybere frontu, do níž bude zpracovaný paket vložen na základě spočtené hashe. Tato akce vyžaduje specifikovat mimo jiné konkrétní hashovací funkci, hashovací klíč a výčet front, do nichž může být paket zařazen. Použití RTE Flow akce RSS v podstatě duplikuje funkcionalitu, která je poskytovaná specifickým programovým rozhraním pro definici RSS. Tato akce rovněž určuje osud paketu.

COUNT Inkrementuje hodnotu čítače paketů, jehož index je specifikován jako argument této akce.

MARK Označí paket hodnotou, která je definována jako argument. Tato hodnota je pak vložena do zvláštní položky v rámci struktury *rte_mbuf* (viz sekce 3.5) a maximální velikost ukládané hodnoty je definována použitou síťovou kartou.

JUMP Přesměruje paket k dalšímu zpracování do skupiny, jejíž index je předán jako argument akce. Pokud v rámci navazující skupiny nedojde k úspěšné klasifikaci, je výsledek nedefinovaný a závislý na konkrétní síťové kartě. Při použití akcí JUMP je třeba dávat pozor, aby nevznikl cyklus při přesměrování mezi skupinami.

AGE Přidává danému pravidlu časovač, který je obnovován průběžně s klasifikovanými pakety. Pokud časovač vyprší (pravidlo neklasifikovalo žádné pakety), je o tomto vyslán speciální signál, pro který lze nastavit obslužnou funkci, která zajistí odstranění pravidla. Jinou možností zjištění pravidel s vypršeným časovačem je použití specializované funkce *rte_flow_get_aged_flows*, která vrátí ukazatele na všechna pravidla a ty je pak možné sekvenčně odstranit.

INDIRECT Speciální typ akce, který může být sdílen mezi více pravidly. Definice těchto pravidel probíhá pomocí specializovaných funkcí.

MODIFY_FIELD Tato akce umožňuje provádět modifikace libovolných políček v hlavičkách paketů. Kromě přímého nahrazení hodnoty za jinou jsou povoleny operace inkrementace a dekrementace.

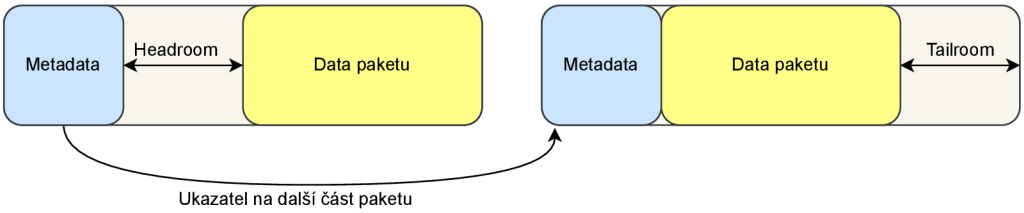
3.5 Specializované knihovny

Framework DPDK poskytuje celou řadu specializovaných knihoven pro jednotlivé fáze zpracování provozu. Tyto knihovny často využívají efektivní algoritmy a vektorizované zpracování. Z těchto knihoven je pro účely optimalizace a akcelerace síťových aplikací možné vybrat zejména:

Správa paketů

Jednotlivé pakety jsou v rámci DPDK aplikace ukládány v rámci struktury *Mbuf* (*message buffer*). Ty jsou definovány v rámci knihovny *rte_mbuf* [2].

(a) Struktura RTE Mbuf pro běžný paket.



(b) Řetězení RTE Mbuf pro uložení *jumbo frame*.

Obrázek 3.3: Znázornění uložení paketu v rámci struktury RTE Mbuf.

Podoba struktury Mbuf je zobrazena na obrázku 3.3. Vidíme, že běžné pakety (velikost menší, než *jumbo frame*) se celé vejdou do jedné struktury (obrázek 3.3a), avšak velké pakety (*jumbo frames*) je nutné rozdělit mezi více instancí struktury a provázat je pomocí ukazatele v hlavičce (metadatech) Mfuf (obrázek 3.3b). Rovněž je vidět, že mezi hlavičkou a samotnými daty paketu je mezera (*headroom*) podobně jako mezi koncem paketu a koncem struktury Mbuf (*tailroom*). *Headroom* je ve struktuře Mbuf umístěn pro zajištění zarovnání dat paketu na velikost řádku paměti cache. *Tailroom* je pak součástí struktury pro její vyplnění na pevnou velikost. Alokace a dealokoace struktury Mbuf je možné provádět efektivně pomocí modulu *memory pool* (sekce 3.2).

Při přijetí paketu ovladač síťové karty extrahuje určité informace do hlavičky Mbufu. Jedná se zejména o hodnotu identifikátoru VLAN, výsledek hashovací funkce RSS, vstupní port a počet zřetězených Mbufů pro konkrétní paket. Důležitou informací uloženou do hlavičky Mbufu je také značka uložená RTE Flow akcí *MARK*. V rámci DPDK aplikace je navíc možné do hlavičky Mbufu umístit další data. K tomuto účelu je v hlavičce definován dynamický prostor (*dynamic space*), v rámci něhož je možné registrovat buď jednobitovou položku (*dynamic flag*), nebo několikabytové políčko (*dynamic field*).

Komponenty pro směrování

DPDK poskytuje dvě oddělené komponenty určené pro směrování provozu. Konkrétně se jedná o komponenty *LPM* a *FIB*. Obě zmíněné komponenty implementují algoritmus *longest prefix match* pro vyhledávání záznamů uložených v interních tabulkách.

Potenciální výhodou komponenty *FIB* je možnost zvolit konkrétní implementaci použitého vyhledávacího algoritmu. Poskytovány jsou však jen dvě varianty: jednoduchý algoritmus *RIB* založený na binárních stromech nebo optimalizovaný algoritmus *DIR-24-8*. Druhý zmiňovaný algoritmus je však rovněž implementován v komponentě *LPM* a rozdíl mezi oběma je tak minimální.

Pro obě komponenty navíc existují dvě další varianty pro pakety IPv4 a IPv6, které poskytují shodné rozhraní, avšak liší se jejich interní implementace algoritmu longest prefix match. Rozdělení mezi dvě další varianty značně komplikuje implementaci jednotné směrovací komponenty, která by pracovala po skupinách paketů (*burstech*), protože je nejprve nutné každý burst rozdělit mezi IPv4 a IPv6 pakety a ty pak zpracovat odděleně.

Hashovací tabulky

Pro ukládání a rychlé vyhledávání jsou frameworkem DPDK poskytovány různé typy hashovacích tabulek. Nejobecnější tabulka je poskytována v knihovně *rte_hash*. Ta dovoluje konfigurovat celkovou velikost hashovací tabulky, velikost uloženého klíče, samotnou hashovací funkci a další specifické parametry. Komponenta pak poskytuje očekávané rozhraní pro vyhledávání, vkládání a mazání záznamů z tabulky.

Zásadní výhodou použití této knihovny je možnost vyhledání skupiny záznamů (*bulk lookup*), při němž je využito vektorových instrukcí procesoru. Další výhodou je možnost souběžného přístupu k tabulce z více vláken.

Obecná klasifikace paketů

Pro klasifikaci paketů poskytuje DPDK rovněž komponentu *ACL* (*Access Control List*). Ta je původně určená k implementaci firewallů, nicméně je možné ji použít k libovolné úloze vyžadující klasifikaci paketů.

Poskytovaným rozhraním vykazuje komponenta použití návrhového vzoru *builder* a příprava celého klasifikátoru je tak rozdělena do tří kroků: inicializace klasifikačního kontextu, přidání pravidel a sestavení datových struktur nutných ke klasifikaci.

Samotná klasifikace paketů pak probíhá po skupinách a je možné ji akcelerovat použitím vektorových instrukcí procesoru. Klasifikaci je také možné provádět souběžně z více vláken, což dále zlepšuje efektivitu této komponenty. Výstupem klasifikátoru je pak vektor čísel, který obsahuje pro každý zpracovaný paket index pravidla, které jej klasifikuje nebo hodnotu 0, pokud paket nebyl klasifikován.

Interně knihovna implementuje algoritmus využívající několik struktur *multibit trie*. Příklad multibit trie je zobrazen na 3.4. Jedná se o n-ární stromovou strukturu uzpůsobenou pro rychlé vyhledávání. Oproti klasické jednobitové variantě trie používá multibit trie více bitů na reprezentaci uzlu, což umožňuje rychlejší průchod. Počet bitů použitý k reprezentaci uzlu se nazývá *stride*. V případě použití multibit trie pro vyhledávání prefixů stride rovněž určuje, o kolik bitů se prefix prodlouží s každou úrovní zanoření do trie. Na obrázku je vidět nastavení hodnoty stride = 2, z čehož vyplývá, že každý uzel obsahuje 2^{stride} ukazatelů na další uzly.

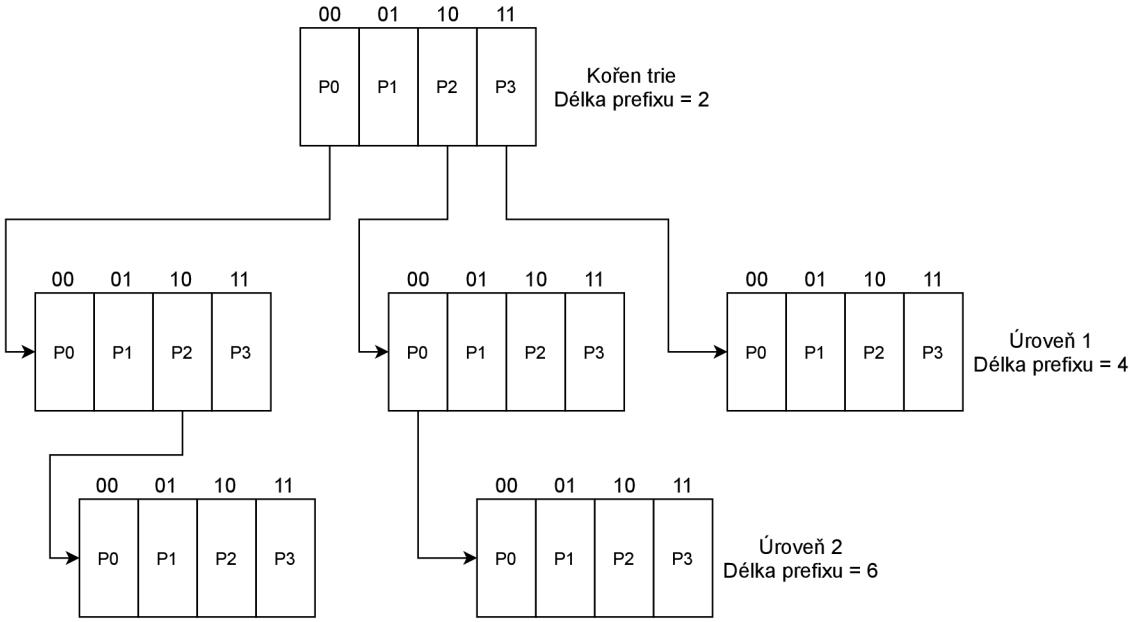
Efektivita algoritmu pro klasifikaci paketů je vyšší, než v případě použití binární varianty trie. Platí, že výška trie je omezena na $\lceil \frac{W}{k} \rceil$, kde W určuje maximální délku vyhledávané položky v bitech (32 pro IPv4, 128 pro IPv6, 48 pro MAC adresy ...) a k odpovídá hodnotě stride. Stride je tedy v tomto případě považován za faktor zrychlení vyhledávání. Na druhou stranu však může růst paměťová náročnost, protože může docházet k expanzi uzlů, které dále nejsou použity [25].

RTE ACL klasifikační algoritmus umožňuje tři běžně používané typy klasifikace:

1. Délka prefixu (v ACL nazývaná jako *MASK*) – používaná typicky pro IP adresy. Pravidla specifikují hodnotu a počet relevantních bitů.

2. Rozsah – obvykle využívaná pro porty transportní vrstvy. Pravidla specifikují spodní a horní mez rozsahu.

3. Bitová maska – používaná například pro identifikátory protokolů. Pravidla určují přesnou bitovou masku určující validní bity.



Obrázek 3.4: Schéma reprezentace multibit trie (se stride = 2) pomocí bloků ukazatelů.

Implementace algoritmu v rámci DPDK je velmi generická a umožňuje klasifikovat položky libovolné velikosti. Existují ovšem určitá omezení na organizace klasifikovaných položek:

- První specifikovaná položka musí mít velikost právě jeden byte.
- Následující položky jsou seskupeny po 4 bytech.

Tato omezení jsou dána z důvodu optimalizace. První jednobytová položka je totiž zpracována jako první a cyklus zpracovávající následující položky je rozbalen a vektorizován pro zvýšení efektivity.

Položky, které jsou menší, než 4 byty (například čísla portů transportní vrstvy) je možné seskupovat po dvojicích nebo přetypovat na čtyřbytovou hodnotu s doplněním nulami. Naopak položky větší, než 4 byty (například IPv6 adresy) mohou být rozděleny do čtyřbytových bloků.

3.6 Nástroj *testpmd*

Nástroj *testpmd* je DPDK aplikace určená k ověřování funkcionality součástí frameworku DPDK z příkazového řádku [9]. Ve výchozím nastavení je nástroj spuštěn v režimu přeposílání paketů na výstup. Je však poskytováno interaktivní konfigurační rozhraní se specifickým příkazovým jazykem. Z pohledu akcelerace DPDK aplikací je nejdůležitější vlastností možnost konfigurovat funkce síťové karty včetně offloadu pomocí RTE Flow [10].

Každý příkaz související s RTE Flow je uvozen klíčovým slovem *flow*. Za tímto úvodem následuje specifikace *operačního režimu*, tedy součásti RTE Flow, která bude konfigurována. Tímto režimem může být například:

1. *info* zjišťuje informace o stavu RTE Flow na konkrétním síťovém portu.
2. *configure* konfiguruje RTE Flow na daném síťovém portu.

3. *create* vytváří RTE Flow pravidlo se specifikovanými parametry.

4. *destroy* odstraní konkrétní RTE Flow pravidlo.

5. *flush* odstraní všechna offloadovaná pravidla.

6. *list* vypíše seznam všech offloadovaných pravidel.

Příklad vytvoření RTE Flow pravidla nástrojem *testpmd* je uveden ve výpisu 3.1. Vidíme, že vytváření pravidla vyžaduje zadat číslo používaného portu (ve výpisu první specifikovaná nula). Dále jsou specifikovány atributy pravidla (v příkladu *group*, *priority* a *ingress*). Následuje definice *matching pattern*, který se v příkladu skládá z položky Ethernet a IPv4. Ethernet v tomto případě nemá definovanou žádnou zdrojovou ani cílovou adresu ani masku, což znamená, že se použije výchozí nastavení uvedené v dokumentaci konkrétního *pattern item* (typicky se jedná o úspěšnou klasifikaci všech paketů). V případě hlavičky IPv4 je definována zdrojová adresa a maska, cílová je opět ponechána ve výchozí konfiguraci. Ve výpisu rovněž vidíme použití dopředných lomítek pro oddělení jednotlivých *pattern items* a ukončení sekce *matching pattern* speciální položkou *end*. Po specifikaci *matching pattern* následuje definice prováděných akcí. V uvedeném výpisu vidíme použití akce *jump* do skupiny 1, za níž následuje ukončovací akce *end*. Po zadání pravidla nakonec nástroj *testpmd* vypíše informaci o úspěšnosti vložení tohoto pravidla do síťové karty (ve výpisu poslední řádek informuje o úspěšném vytvoření).

```
testpmd> flow create 0 group 0 priority 1 ingress \
    pattern eth / ipv4 src spec 10.0.0.0 src mask 255.255.0.0 / end \
    actions jump group 1 / end
Flow rule #0 created
```

Výpis 3.1: Příklad vytvoření RTE Flow pravidla nástrojem *testpmd*.

Kapitola 4

Aplikace pro potlačení DDoS útoků

Aplikace *DDoS Protector* pro potlačení distribuovaných útoků typu odepření služeb je vyvíjená v rámci sdružení CESNET. Aplikace je implementována v jazyce C s využitím frameworku DPDK. Obrázek 4.1 zobrazuje vysokoúrovňové schéma aplikace DDoS Protector. Vidíme, že aplikace je rozdělena do dvou zřetězených linek – *processing pipeline* a *system interface pipeline* mezi nimiž je možné komunikovat pomocí DPDK struktur typu *ring* [7].

Processing Pipeline

V horní části obrázku 4.1 je oranžovou barvou znázorněna zřetězená linka *Processing Pipeline*, která je zodpovědná za samotné zpracování paketů. Tento úkol je výpočetně náročný, proto je linka rozdělena mezi N vláken, které provoz zpracovávají paralelně. Z hlediska modelů paralelních DPDK aplikací (sekce 3.3) se ale stále jedná o model *run-to-completion*, protože každé vlákno pracuje s celou pipeline.

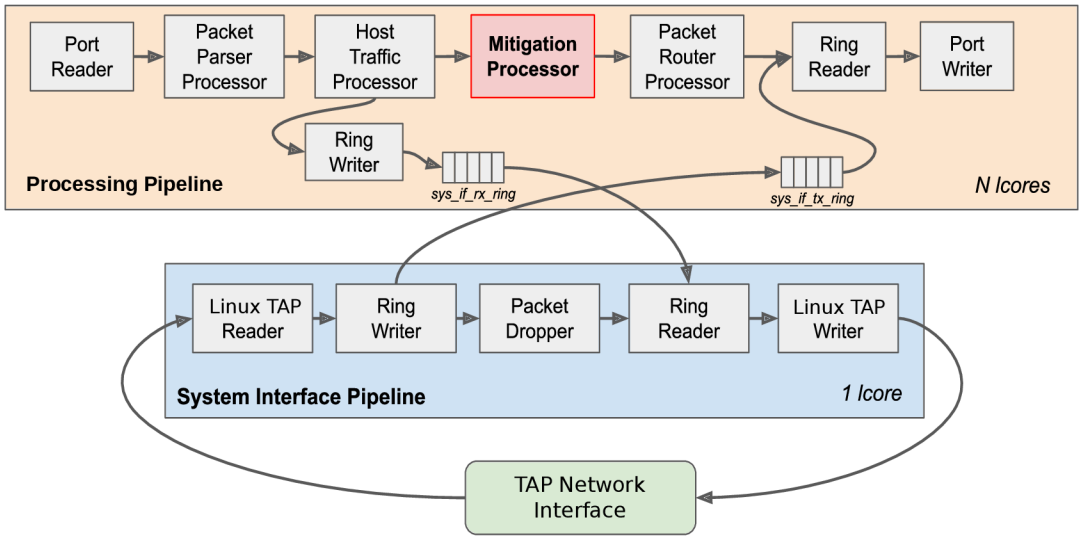
Pipeline se skládá z následujících komponent:

Port Reader Tato komponenta provádí konfiguraci vstupního rozhraní síťové karty a poskytuje nad tímto rozhraním vrstvu abstrakce. Port reader může rovněž konfigurovat odstraňování VLAN hlaviček (*VLAN stripping*) a filtraci paketů na základě jejich MAC adresy. Obě zmíněné operace také mohou být offloadovány přímo na síťovou kartu pomocí rozhraní RTE Flow.

Packet Parser Processor Jedná se o komponentu zodpovědnou za extrakci hlaviček paketu a ověření jejich správnosti a integrity. Extrahované hlavičky pak mohou být vloženy jako dynamická položka uvnitř samotného *rte_mbuf* (sekce 3.5). Díky tomuto uložení pak navazující komponenty nemusejí hlavičky extrahovat znovu.

Host Traffic Processor Tato komponenta je zodpovědná za identifikaci provozu, který je směrován přímo aplikaci DDoS Protector. Takto identifikované pakety jsou pak předány komponentě *Ring Writer*, která následně zprostředkuje komunikaci se *System Interface Pipeline*.

Komponenta Host Traffic Processor je v aktuální implementaci akcelerována offloadem směrovacích pravidel pomocí knihovny RTE Flow. Síťová karta tak identifikuje pakety sama a pomocí akce *MARK* je označí. V software pak již není třeba pakety znovu identifikovat a je pouze přečtena značka uložená v paketu.



Obrázek 4.1: Schéma aplikace DDoS Protector.

Mitigation Processor Tato komponenta realizuje klíčovou funkcionalitu celkové aplikace, a to mitigaci DDoS útoků. K tomuto účelu je implementováno několik modulů, které jsou popsány v sekci 4.1. Celkový pohled na samotnou komponentu je pak podrobněji popsán v sekci 4.3.

Packet Router Processor Jedná se o komponentu, která provádí směrování paketů na základě uložené směrovací tabulky. Pro korektní směrování je potřeba vyhledat odpovídající záznam s informací o MAC adrese a VLAN, které budou nastaveny odchozímu paketu. Kromě toho je navíc provedeno snížení hodnoty *Time To Live* (*TTL*) v hlavičce IP.

Interně je pro implementaci použito dvou DPDK komponent typu LPM a pakety jsou tak nejprve rozděleny mezi IPv4 a IPv6 a tyto skupiny jsou poté zpracovány odděleně. Po zpracování je skupiny nutné znovu sloučit.

Port Writer Stará se o konfiguraci výstupního portu síťové karty a poskytuje abstrakci pro odesílání paketů. Odesílané pakety mohou pocházet jak z předchozí komponenty *Packet Router Processor*, zároveň ale mohou být přijímány z druhé zřetězené linky *System Interface Pipeline*. Komunikaci mezi oběma linkami v tomto případě zajišťuje *Ring Reader*, který abstrahuje DPDK komponentu *rte_ring*.

Port Writer provádí rovněž vložení nové VLAN hlavičky do paketu, což může být také offloadováno přímo do HW síťové karty.

System Interface Pipeline

Ve spodní části obrázku 4.1 je modrou barvou znázorněna linka *System Interface Pipeline*, jejímž cílem je vytvářet a spravovat síťové rozhraní operačního systému. Takto vytvořené rozhraní zpřístupňuje fyzické rozhraní síťové karty a umožňuje fyzické rozhraní konfigurovat

a také pomocí něj komunikovat. Protože předpokládáme, že množství provozu na této lince je podstatně nižší než v Processing Pipeline, je pro tuto linku vyhrazeno pouze jedno vlákno.

Komponenty ve schématu 4.1 označené jako *Linux TAP Reader* a *Linux TAP Writer* odpovídají dříve zmíněným komponentám *Port Reader*, respektive *Port Writer*. V tomto případě však nereprezentují fyzický port síťové karty, ale softwarové rozhraní *TAP* [14].

Historickou alternativou použití TAP rozhraní byl speciální modul jádra *KNI* (*Kernel NIC Interface*) [5]. Tato komponenta však vykazovala chyby a byla proto nahrazena.

4.1 Mitigační moduly

Hlavní funkcí aplikace DDoS Protector je provádět mitigaci různých typů DDoS útoků (kapitola 2). K tomuto účelu jsou implementovány nezávislé mitigační moduly. Těchto modulů může být zároveň aktivních několik. Jednotlivé mitigační moduly můžeme rozdělit na pasivní a interaktivní. Pasivní moduly jsou založeny na analýze příchozího síťového provozu a zahazování paketů. Interaktivní metody navíc ještě mohou generovat podvržené pakety, které jsou zaslány zpět na stranu klienta za účelem ověření jeho legitimity.

Pro přidávání nových mitigačních modulů poskytuje aplikace DDoS Protector specializované rozhraní a díky tomu je možné snadno přidávat nové moduly. Aktuálně jsou implementovány tyto mitigační moduly:

Mitigace amplifikačních útoků Metoda mitigace amplifikačních útoků si uchovává tabulku IP adres, ze kterých pochází největší podíl příchozích paketů. Při překročení nastaveného prahu objemu příchozích dat (paketů nebo bitů) je pak provoz z nejaktivnějších IP adres zahazován. Takto je možné zabránit zahlcení chráněných zařízení nebo síťových linek.

SYN Drop [19] Tato pasivní mitigační metoda chrání před útoky záplavou TCP SYN paketů. SYN Drop metoda uchovává tabulku, v niž přiřazuje jednotlivým prefixům zdrojových sítí počet přijatých SYN paketů a ACK paketů. O zahazování paketů je pak rozhodováno na základě dvou prahových hodnot. Tzv. *soft threshold* určuje počet SYN paketů, které je možné propustit, aniž by byly přijaty odpovídající ACK pakety. Druhá z prahových hodnot, tzv. *hard threshold*, pak určuje počet SYN paketů, které je možné přijmout nezávisle na počtu přijatých ACK paketů. *Soft threshold* tak určuje, kolik částečně ustavených spojení může být naráz otevřeno, aniž by byly pakety zahazovány.

Další vlastností SYN Drop metody je automatické zahození prvního přijatého SYN paketu každého spojení. To nepůsobí žádné problémy v případě legitimního provozu, kdy je jednoduše SYN paket zaslán znovu. Zároveň ale tato technika zabrání útočníkovi zasílat velké množství paketů s náhodně podvrženou IP adresou.

TCP Autentizace [19] Tato interaktivní metoda je určená k mitigaci útoků typu SYN Flood. Principem je zahození prvního SYN paketu a autentizace spojení na základě podvržení TCP segmentu s nastavenými příznaky SYN a ACK a hlavičkou *acknowledge number* nastavenou na dosud nepoužitou hodnotu. Takto nastavené *acknowledge number* je v rámci TCP komunikace považováno za chybu a klient by měl odpovědět TCP segmentem s příznakem RST a shodně nastaveným *acknowledge number*. Pokud je takový paket přijat, je klient úspěšně autentizován a jeho následující komunikace může probíhat bez omezení.

Legitimní klient by touto komunikací neměl být omezen, protože operační systémy zpravidla pokus o navázání spojení opakují. Naopak u SYN Flood útoků jsou často generovány pouze segmenty s příznakem SYN, k autentizaci nikdy nedojde a pakety jsou opakovaně zahazovány.

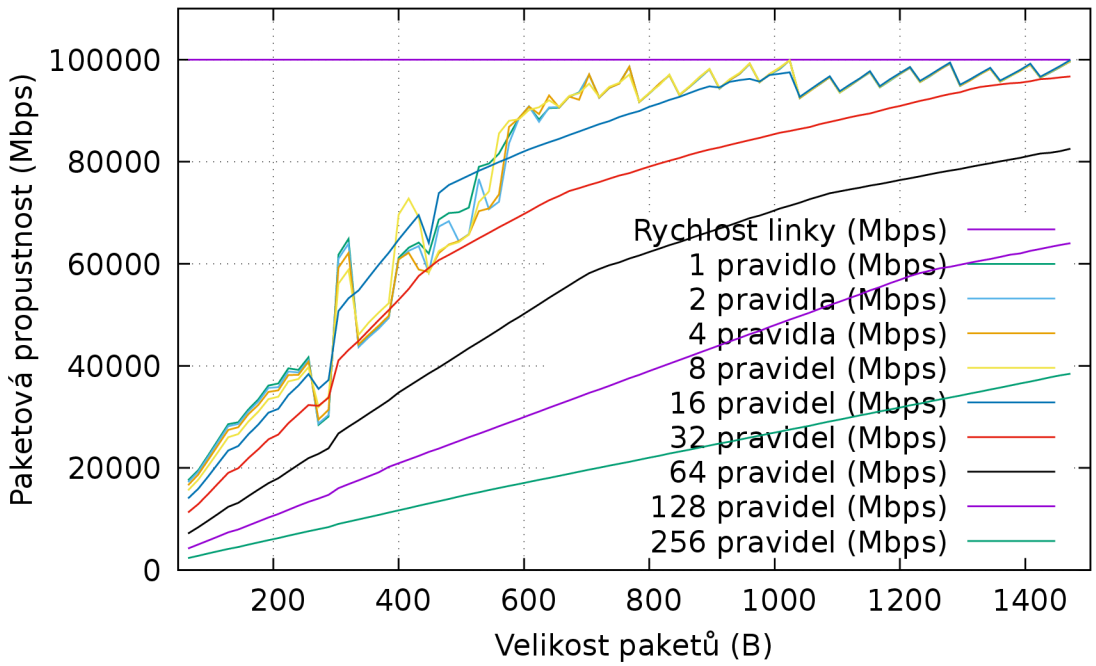
4.2 Analýza výkonu aplikace

Pro určení součástí aplikace, které je vhodné optimalizovat bylo provedeno několik kroků analýzy. Nejprve byla provedena analýza aplikace pomocí testovacího prostředí využívajícího nástroj *Cisco Trex* [26], díky které byla zjištěna propustnost aplikace. Graf naměřené propustnosti v závislosti na velikost paketů je zobrazen na obrázku 4.2.

Propustnost byla měřena s různým počtem pravidel nahraných do mitigačního jádra, což je v grafu zobrazeno čarami různých barev. Vidíme, že propustnost aplikace se se zvyšujícím počtem pravidel snižuje a už při 256 pravidlech se ani nepřibližuje rychlosti linky 100 Gb/s. Tento jev je nežádoucí a ukazuje na nevhodnou implementaci mitigačního jádra.

Zmíněnou závislost pak dokazuje profilování pomocí nástroje *perf*, v rámci něhož byl analyzován čas strávený v jednotlivých částech programu DDoS Protector. Výsledky tohoto profilování po klíčových komponentách jsou zobrazeny v tabulce 4.1. Z této tabulky je zřejmé, že mitigační jádro zabírá samo o sobě více než polovinu času aplikace. Grafické znázornění času stráveného jedním procesorovým jádrem v jednotlivých funkcích je pak zobrazeno na obrázku 4.3. Toto zobrazení využívá metodu *flamegraph* [21]. Flamegraph zobrazuje hierarchii volaných funkcí v programu na ose y. Osa x pak znázorňuje dobu, kterou zabírá volaná funkce v rámci doby běhu volající funkce.

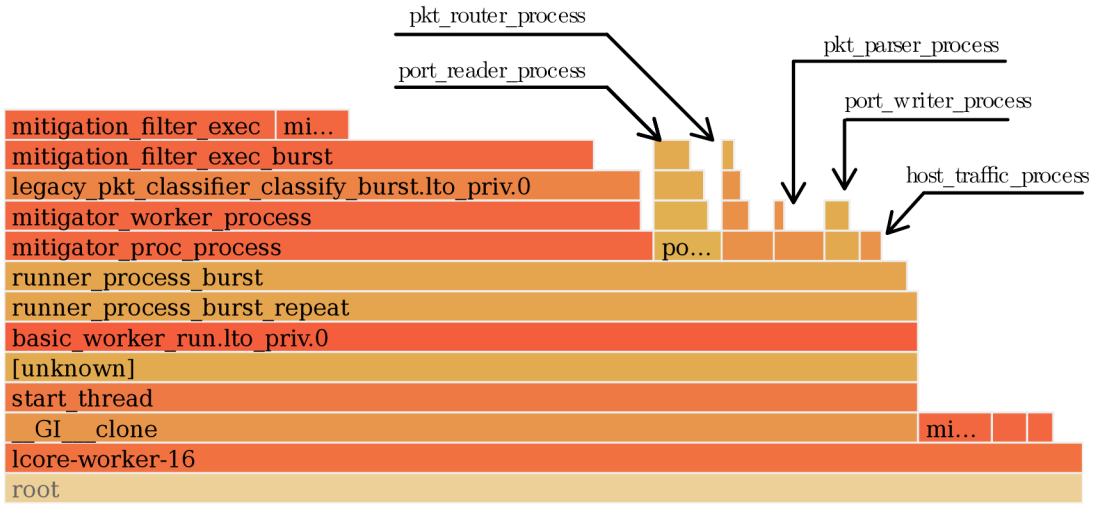
Uvedený flamegraph ukazuje, že nejnáročnější funkcí v rámci mitigačního jádra je *legacy_pkt_classifier_classify_burst*, která provádí klasifikaci paketů po burstech. Klasifikaci paketů v rámci mitigačního jádra se podrobněji věnuje sekce 4.4.



Obrázek 4.2: Bitová propustnost aplikace DDoS Protector.

Komponenta	Relativní čas [%]
Mitigation Processor	59.633
Port Reader	6.808
Packet Router Processor	4.581
Packet Parser Processor	4.305
Port Writer	4.188
Host Traffic Processor	1.769

Tabulka 4.1: Analýza výkonnosti jednotlivých komponent při použití 256 mitigačních pravidel.



Obrázek 4.3: Zastoupení jednotlivých funkcí v celkové době běhu programu DDoS Protector zobrazené metodou *flamegraph*.

4.3 Mitigační jádro

Mitigační jádro tvoří společnou část pro jednotlivé mitigační metody (sekce 4.1) a poskytuje rozhraní pro implementaci těchto metod. Mitigační jádro samotné na základě definovaných pravidel zpracovává síťový provoz a získává z něj statistky, které dále předává připojeným mitigačním modulům. V případě probíhajícího útoku smí blokovat část provozu, aby došlo ke snížení objemu provozu směřujícího do cílové sítě útoku. Tímto způsobem je pak zajištěna mitigace samotného útoku.

Konfigurace mitigačního jádra se opírá o specifikaci pravidel. Každé pravidlo se skládá ze dvou hlavních částí. Společná část (identifikátor VLAN, zdrojové a cílové sítě, rozsahy zdrojových a cílových portů transportní vrstvy) je zpracovávána samotným mitigačním jádrem. Specifická část je pak zpracovávána příslušným mitigačním modulem.

Pravidla pro jednotlivé mitigační metody pak mohou obsahovat práh (threshold), který definuje počet paketů nebo bytů za sekundu, při jejichž překročení dojde k zahájení potlačení provozu v daném síťovém toku. Pravidlo, u nějž došlo k překročení prahové hodnoty mitigace, se pak považuje za aktivní. V opačném případě je pak pravidlo neaktivní a daný tok není omezen.

Architektura mitigačního jádra se opírá o klíčové body:

1. Periodické střídání dvou *mitigačních oken*, kdy jedno okno je považováno za aktivní a jejichž výměna probíhá bez přerušení zpracování paketů.

2. Použití jednoho řídicího *master* vlákna a několika *worker* vláken. Jednotlivá *worker* vlákna zpracovávají provoz v rámci aktivního okna, zatímco *master* vlákno počítá statistiky z uplynulého okna.

3. Atomická změna pravidel a jejich datových struktur za běhu. Tohoto je možné dosáhnout díky použití dvou mitigačních oken.

4. Využití několika mitigačních metod, které jsou nezávislé (viz sekce 4.1).

Samotné zpracování paketů probíhá v časových úsecích (oknech). Každé worker vlákno si tak ukládá pravidla a datové struktury, které jsou platné jen po dobu trvání daného okna a po jeho skončení jsou nahrazeny novými. Prakticky jsou tak implementovány dvě sady datových struktur – aktivní a neaktivní. To sice vede k větší spotřebě paměti, nicméně dovoluje to snáze realizovat atomickou výměnu těchto datových struktur. Do aktivních struktur jsou ukládány nové informace z aktuálního mitigačního okna a zároveň probíhá vyhodnocení zpracovaných dat z neaktivních struktur. Na základě takto zpracovaných dat pak mohou být aktivovány nebo deaktivovány určité mitigace v následujícím okně.

Hlavní smyčka řídící běh mitigátoru a výměnu mitigačních oken se skládá z:

1. Fáze *swap* na začátku každé iterace provede výměnu aktivních mitigačních oken.

2. Fáze *evaluate* dále zahájí vyhodnocení veškerých dat zaznamenaných v uplynulém okně a rozhodne o aktivaci mitigačních metod.

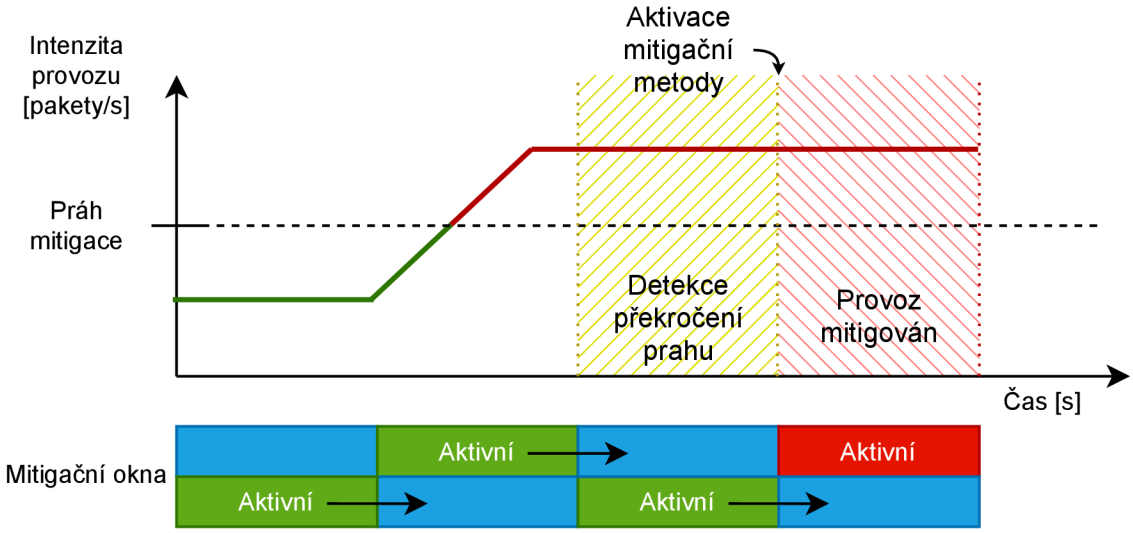
3. Fáze *reload* nakonec zkontroluje, zda nedošlo ke změně instalovaných pravidel. V případě změny pak inicializuje datové struktury pro novou sadu pravidel.

Obrázek 4.4 zobrazuje časový průběh aktivace mitigační metody pro konkrétní síťový tok. Vidíme, že ve druhém okně dochází k překročení prahové hodnoty, což je pak detekováno v následujícím okně. V něm pak také dochází ke konfiguraci posledního okna tak, aby mohla být realizována samotná mitigace. Po poslední výměně oken běží tedy jednotlivá vlákna v režimu potlačení tohoto síťového toku, což je na obrázku znázorněno červeným zvýrazněním aktivního okna.

4.4 Paketový klasifikátor

Jak bylo zmíněno v předchozí sekci, pravidla určená pro mitigátor obsahují společnou část, díky které je možné určit příslušnost paketu k síťovému toku na základě hodnot v jeho hlavičce. K tomuto účelu existuje v mitigačním jádře komponenta zvaná *filtr* implementující klasifikaci paketů. Tato komponenta je zařazena před všemi ostatními mitigačními moduly a musí zpracovat všechny příchozí pakety. Z toho vyplývá, že efektivita použitého klasifikačního algoritmu (a tedy efektivita filtru) velmi ovlivňuje výkon celého mitigátoru.

Filtr implementovaný v současném mitigátoru funguje na principu lineárního průchodu seznamem pravidel. Pro každé pravidlo pak zjišťuje, které pakety z aktuální skupiny (burstu) jsou tímto pravidlem klasifikované. Z toho plyne, že jeden paket může být klasifikován více pravidly. Důležitou vlastností je, že jedno pravidlo filtru může být společné

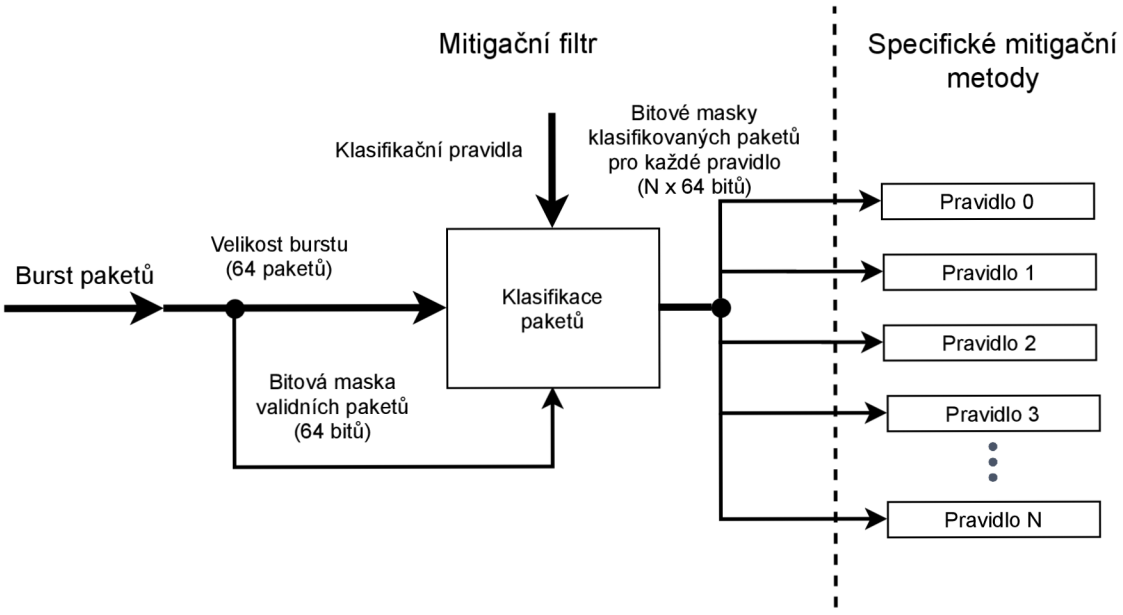


Obrázek 4.4: Schéma aktivace mitigační metody při překročení prahové hodnoty intenzity provozu.

pro několik mitigačních modulů. Každý paket, který je klasifikován filtrem je tedy nutné postoupit dále ke zpracování všem navazujícím modulům.

Obrázek 4.5 ukazuje schéma zpracování v rámci mitigačního filtru. Je vidět, že pakety jsou na vstupu přijímány po burstech. Výchozí velikost burstu je nastavena na hodnotu 64. Z tohoto burstu mohou být navíc některé pakety považovány za nevalidní. Abychom mohli určit, které pakety z daného burstu jsou určeny ke zpracování, je použita bitová maska, kde hodnota 1 značí, že paket je validní, a tedy má být zpracován. Velikost této bitové masky odpovídá velikosti burstu (implicitně tedy 64 bitů). Validní pakety jsou pak klasifikovány pomocí nahraných klasifikačních pravidel. Výstupem konkrétního klasifikačního algoritmu je pak sada bitových masek (pro každé mitigační pravidlo jedna). Ty pak určují, které pakety burstu byly klasifikovány daným pravidlem. Na základě této informace jsou potom volány specifické mitigační metody jednotlivých pravidel, což už však nespadá do kompetence mitigačního filtru.

Životní cyklus filtru jakožto komponenty je omezen platností pravidel. Vždy když jsou v mitigátoru změněna klasifikační pravidla, musí dojít k nahrazení existující instance filtru za novou na základě aktualizovaných pravidel. Toto je realizováno opět pomocí mitigačních oken. Zde ale s rozdílem, že filtr je sdílen oběma okny. Tedy při požadavku na změnu pravidel dojde k alokaci nových struktur filtru, které však nejsou předány žádnému z mitigačních oken, a tedy nový filtr není aktivní. Až v okamžiku výměny oken je provedeno atomické přiřazení nové instance filtru. Původní instance je následně dealokována.



Obrázek 4.5: Schéma zpracování burstu paketů mitigačním filtrem.

Kapitola 5

Návrh optimalizace a akcelerace

Aplikaci DDoS Protector je možné akcelerovat jak na úrovni software použitím efektivnějších algoritmů, tak na úrovni hardware pomocí offloadu výpočtu na síťovou kartu. Z naměřených výsledků profilování vyplývá, že komponenta mitigátoru vykazuje závislost propustnosti na počtu pravidel. Tato závislost se projevuje již při nízkém počtu pravidel a je způsobena implementací paketového klasifikátoru. Prvním krokem optimalizace je tedy nahrazení softwarového klasifikačního algoritmu. Takto upravený paketový klasifikátor by pak mohl sloužit jako základ pro hardwarovou akceleraci pomocí offloadu klasifikačních pravidel knihovnou RTE Flow.

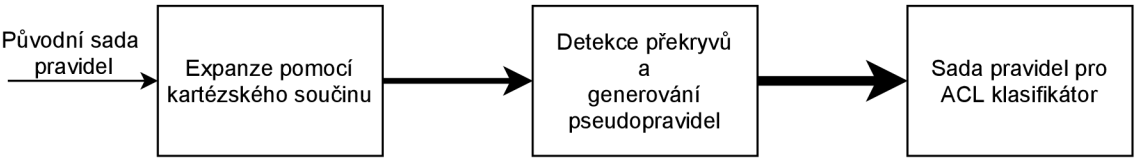
5.1 Nahrazení klasifikačního algoritmu

Efektivních alternativ k současnému klasifikačnímu algoritmu je možné v literatuře nalézt mnoho [24]. Některé jsou vhodné pro softwarové zpracování paketů, některé jsou použitelné spíše pro hardware. Přímo ve frameworku DPDK je však poskytnuta knihovna *RTE ACL* [6], která poskytuje relativně jednoduché rozhraní pro klasifikaci paketů.

Pro možnost nahrazení klasifikačního algoritmu uvnitř mitigačního filtru je nutné zajistit, aby jeho funkčnost odpovídala existující variantě. Hlavními požadavky, které RTE ACL klasifikátor oproti existující variantě implicitně neumožňuje, jsou podpora seznamů sítí nebo rozsahů portů a možnost klasifikovat každý paket více pravidly současně.

Z tohoto důvodu bude potřeba provádět předzpracování sady pravidel v několika stupních. To je zobrazeno na blokovém schématu 5.1. V tomto schématu je také tloušťkou šipky znázorněno, že každý krok předzpracování potenciálně zvětšuje instalovanou sadu pravidel.

Jednou z vlastností, kterou je nutné dodržet je převod formátu pravidel poskytovaného mitigačním jádrem na formát vyžadovaný knihovnou RTE ACL. Takový převod je zobrazen tabulkou 5.1. Vidíme, že původní pravidlo obsahovalo seznam dvou zdrojových IP sítí a seznam dvou rozsahů cílových portů. Klasifikátor poskytovaný knihovnou RTE ACL však nepodporuje seznamy položek v pravidlech a je tedy nutné provést kartézský součin



Obrázek 5.1: Schéma předzpracování sady pravidel pro ACL klasifikátor.

ID	1
VLAN	100
Source IP	10.0.0.14/32 10.0.1.0/24
Destination IP	192.168.14.0/24
Source Port	0 – 65535
Destination Port	22 80 – 89
L4 Protocol	TCP UDP

(a) Příklad pravidla použitého mitigačním jádrem.

ID	2	ID	3
VLAN	100	VLAN	100
Src IP	10.0.0.14/32	Src IP	10.0.1.0/24
Dst IP	192.168.14.0/24	Dst IP	192.168.14.0/24
Src Port	0 – 65535	Src Port	0 – 65535
Dst Port	22	Dst Port	22
L4 Proto	TCP UDP	L4 Proto	TCP UDP

ID	4	ID	5
VLAN	100	VLAN	100
Src IP	10.0.0.14/32	Src IP	10.0.1.0/24
Dst IP	192.168.14.0/24	Dst IP	192.168.14.0/24
Src Port	0 – 65535	Src Port	0 – 65535
Dst Port	80 – 89	Dst Port	80 – 89
L4 Proto	TCP UDP	L4 Proto	TCP UDP

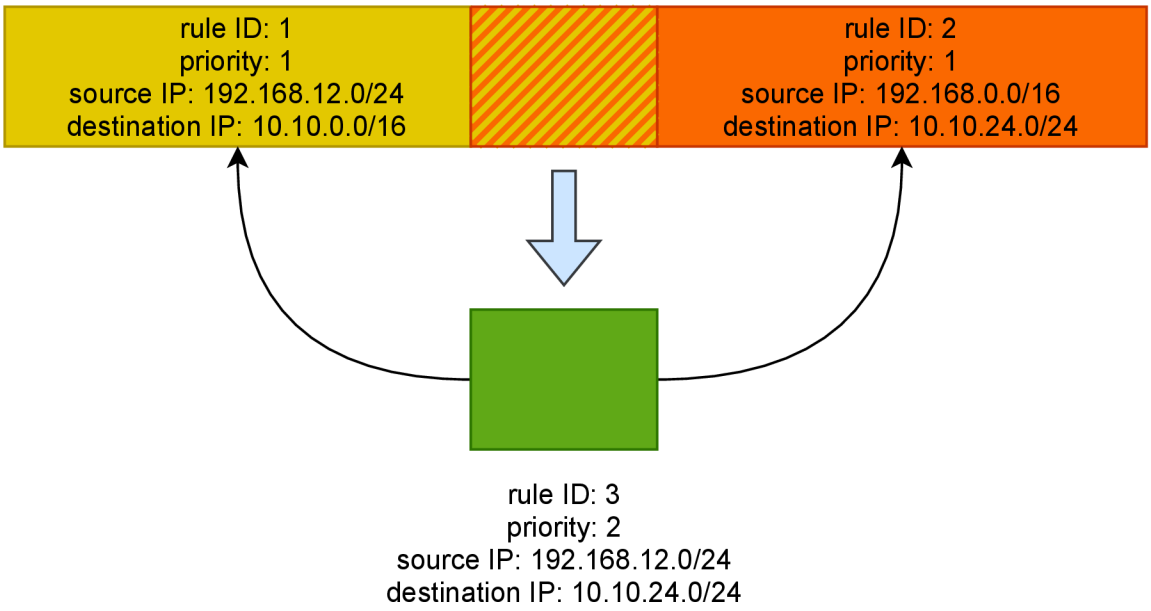
(b) Expanze pravidla pomocí kartézského součinu vybraných položek.

Tabulka 5.1: Příklad odstranění seznamů sítí a seznamů rozsahů portů pomocí vytvoření nových pravidel.

mezi seznamy *Source IP*, *Destination IP*, *Source Port* a *Destination Port*. Z této expanze ve zmíněném příkladu vzniknou celkově 4 pravidla.

Ve stejné tabulce je také vidět, že pravidla obsahují více protokolů vrstvy L4 zároveň. V tomto případě však není nutné provádět kartézský součin, jelikož se jedná o bitovou masku, se kterou umí ACL klasifikátor zacházet.

Další vlastností, kterou musí optimalizovaný mitigační filtr splňovat je možnost klasifikovat jeden paket více pravidly. Tohoto jevu však docílíme jen tehdy, když mezi klasifikačními pravidly existují překryvy nebo alespoň částečné průniky. Tuto situaci RTE ACL neumí řešit samo o sobě. Pokud je totiž specifikována datová sada s překryvy nebo průniky, které mají stejnou prioritu, výsledek klasifikace pro daný paket není definován. Řešením této situace je použití adaptéru, který nejprve provede předzpracování sady pravidel a pak i převod výsledků klasifikace do podoby očekávané mitigačním filtrem.



Obrázek 5.2: Vytvoření nového pseudopravidla na základě dvou překrývajících se pravidel.

Předzpracování pravidel může mít podobu rozgenerování tzv. *pseudopravidel*. Pseudopravidlem rozumíme pravidla, která jsou sice použita pro klasifikaci paketů v rámci RTE ACL klasifikátoru, neodpovídají však přímo žádnému z pravidel zadaných uživatelem.

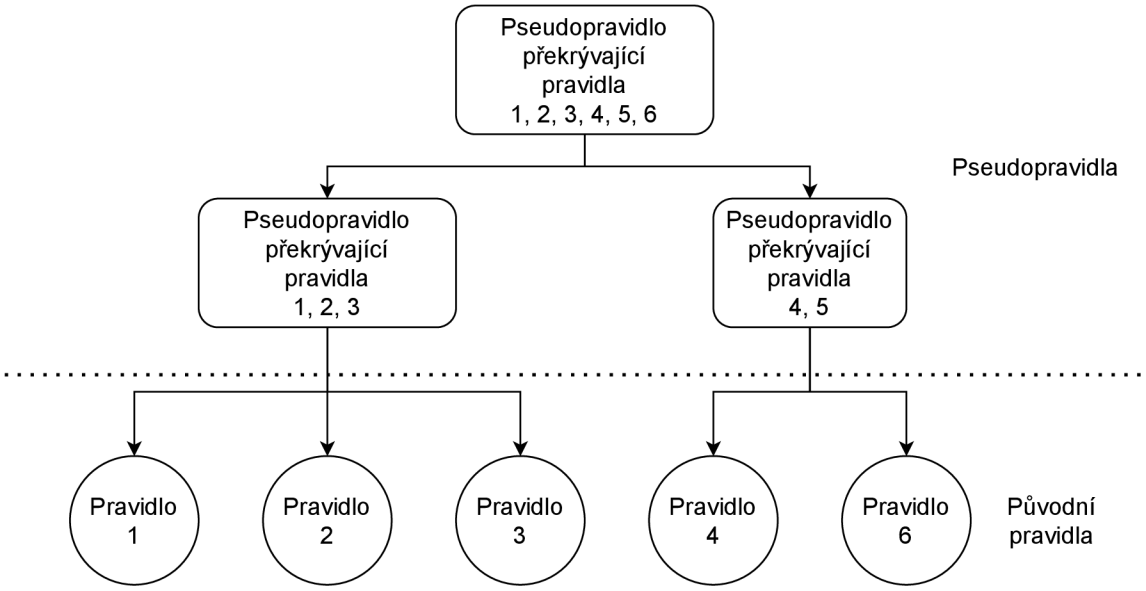
Jednoduchý příklad generování pseudopravidla můžeme vidět na obrázku 5.2. Vidíme, že překryv nastává, protože zdrojová IP adresa pravidla 2 je obecnější než u pravidla 1 a naopak cílová IP adresa pravidla 1 je obecnější než u pravidla 2.

U skutečných mitigačních pravidel je třeba řešit ještě překryvy v rámci rozsahů portů transportní vrstvy. Řešením této situace je najít průnik mezi těmito pravidly a vytvořit z něj pravidlo nové (pseudopravidlo), kterému je přiřazena vyšší priorita než oběma existujícím pravidlům. To zajistí přednostní klasifikaci pseudopravidly. Nové pseudopravidlo pak také obsahuje odkazy na původní pravidla. Při následném zpracování výsledků klasifikace tak bude možné určit všechna pravidla, která paket klasifikovala.

Z této situace je zřejmé, že mezi pravidly a pseudopravidly vzniká stromová hierarchie, která je zobrazena na obrázku 5.3. Pravidla v nejnižších (listových) úrovních stromu reprezentují pravidla původně specifikovaná uživatelem a vyšší úrovně pak reprezentují pseudopravidla. Nová pseudopravidla tedy mohou vznikat i z již vytvořených pseudopravidel.

Předzpracování pravidel by mělo probíhat ve fázi přípravy klasifikační komponenty, tedy v neaktivním mitigačním okně (sekce 4.3). Časová náročnost tohoto předzpracování pravděpodobně nebude mít zásadní vliv na výkon samotné aplikace. I přesto je však třeba zajistit nahrání pravidel v čase, který by neměl překračovat hranice mitigačního okna.

Naivním přístupem pro generování pseudopravidel by mohlo být porovnávání každého pravidla s každým a hledání překryvů a průniků mezi nimi. Tento způsob pravděpodobně bude velmi neefektivní. Daleko efektivnější je provést generování po částech. Velmi snadno lze provést rozdělení celé sady pravidel do menších skupin, které vždy odpovídají jedné hodnotě VLAN identifikátoru a v expanzi pravidel provádět v rámci těchto skupin. Dělení do skupin je možné provádět i na úrovních prefixů zdrojových a cílových adres. Z tohoto rozdělení opět vznikne stromová struktura, kde bude relativně snadno možné určit, které



Obrázek 5.3: Schéma stromové hierarchie pseudopravidel.

konkrétní pravidla mezi sebou mohou mít nějaký překryv nebo průnik a nová pseudopravidla generovat pouze z nich.

Naopak zpracování výsledků vrácených z RTE ACL klasifikátoru a jejich převod do podoby použitelné v rámci mitigátoru (sestavení bitových masek pro každé pravidlo, viz sekce 4.4) probíhá s každým příchozím burstem paketů, a tedy je potřeba, aby probíhalo rychle.

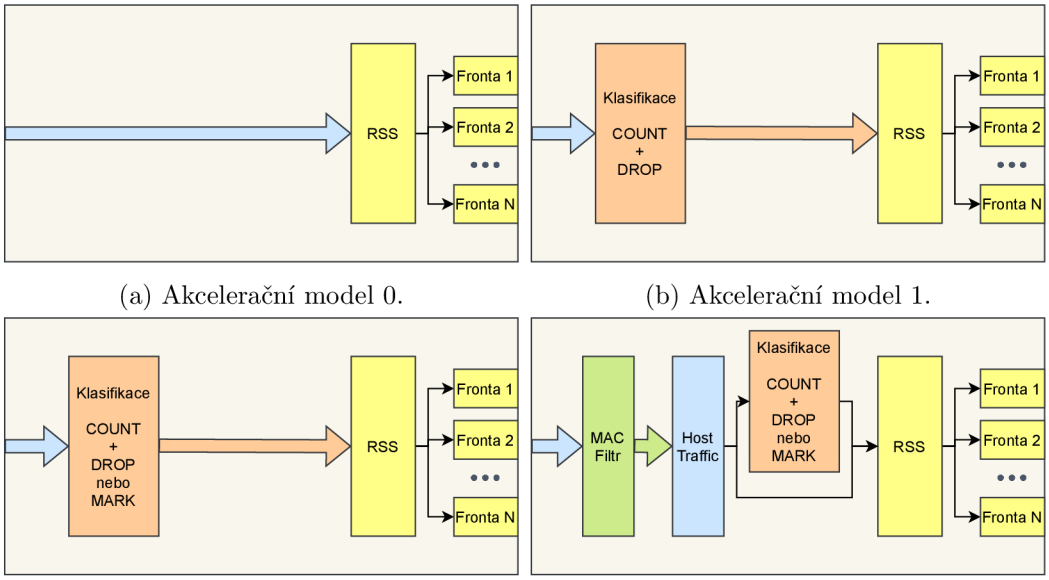
Jelikož výstupem RTE ACL klasifikátoru je vektor čísel, která určují ID použitých klasifikačních pravidel (ta se mohou lišit po předzpracování od původních mitigačních pravidel), je třeba lineárně projít tento vektor a převést každé ID na jedno nebo více ID původních mitigačních pravidel. K tomuto účelu je možné využít stromovou hierarchii pravidel a pseudopravidel, a tedy použít každý prvek výstupního vektoru jako kořen stromu a procházet až na listovou úroveň, která určuje ID původních mitigačních pravidel.

5.2 Možnosti využití RTE Flow

Knihovna RTE Flow poskytuje celou řadu možností, jak přesunout část výpočtu do hardware. Bohužel, podpora RTE Flow se na jednotlivých kartách liší. Bylo proto identifikováno několik modelů offloadu výpočtu [27], které jsou znázorněny na obrázku 5.4.

Model 0 Síťové karty tohoto typu zpravidla nepodporují RTE Flow a hardwarová akcelerace je provedena pomocí specializovaného rozhraní. Tento model je znázorněn na obrázku 5.4a. Na těchto kartách je zpravidla offloadována jen operace RSS. Alternativně je možné, že karta podporuje jen jedno RTE Flow pravidlo, které se však ovladačem přeloží na stejnou konfiguraci, která by byla provedena specializovaným rozhraním.

Model 1 Tento model je zobrazen na obrázku 5.4b. Tyto síťové karty sice podporují klasifikaci paketů, ale podporují pouze akce *COUNT* a *DROP*. Díky těmto dvěma akcím je možné realizovat mitigaci určitého provozu a zároveň počítat zahozené pakety ke zjištění úspěšnosti mitigace.



(a) Akcelerační model 0. (b) Akcelerační model 1.

(c) Akcelerační model 2. (d) Akcelerační model 3.

Obrázek 5.4: Modely akcelerace aplikace DDoS Protector pomocí RTE Flow.

Model 2 Model 2 je zobrazen na obrázku 5.4c. Jedná se o síťové karty, které navíc podporují akci *MARK*. Díky tomu je možné značkovat síťové toky, což následně ulehčí práci mitigačním modulům v software.

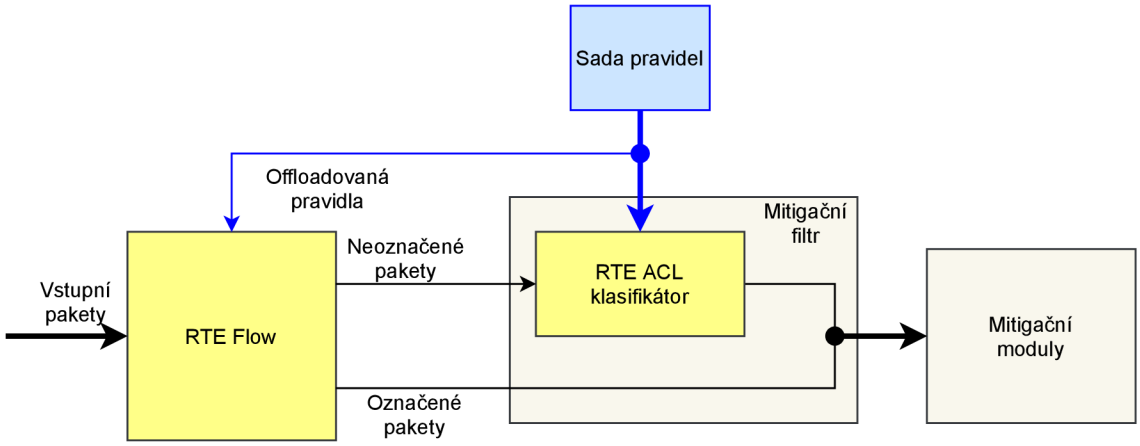
Model 3 Tento komplexnější model je znázorněn obrázkem 5.4d. Model 3 vyžaduje, aby síťová karta umožňovala specifikovat více skupin pravidel a v nejlepším případě i více priorit. Díky tomu je možné akceleraci provádět ve více stupních. První skupina tak může akcelerovat komponentu *Host Traffic* (viz kapitola 4) pomocí značkování paketů určených operačnímu systému. Následující skupiny jsou pak schopné provádět akceleraci mitigací. Poslední skupina nakonec provádí RSS. Pokud je žádoucí provádět filtraci MAC adres pomocí RTE Flow, pak je možné ji předřadit akceleraci *Host Traffic*. Tento typ akcelerace je zobrazen na obrázku 5.4d.

5.3 Offload klasifikačních pravidel

Knihovna RTE Flow umožňuje realizovat přesun klasifikace paketů mimo procesor do samotné síťové karty. Toto řešení umožňuje akcelerovat výpočet pomocí dedikovaných obvodů na čipu síťové karty.

Pro akceleraci mitigačního filtru pomocí RTE Flow je možné vytvořit sadu pravidel, která realizuje klasifikaci. Problematický však může být výstup klasifikace, jelikož veškeré zpracování probíhá v hardware. Mitigační jádro může totiž obsahovat několik mitigačních modulů, které jsou aktivní současně a každý modul musí pakety zpracovat individuálně. Pokud by tedy veškerá mitigace probíhala v hardware (pomocí akce *DROP*), pak by některé moduly nemusely obdržet informaci o úspěšné klasifikaci paketu a nedošlo by k aktualizaci statistik jednotlivými moduly.

Nicméně, RTE Flow poskytuje také akci *MARK*, která do zvláštního pole struktury *Mbuf* (viz sekce 3.5) vloží předem definovanou hodnotu (značku). V tomto případě může být značkou ID mitigačního pravidla, kterému RTE Flow pravidlo odpovídá. Mitigační filtr



Obrázek 5.5: Schéma použití RTE Flow klasifikátoru při nemožnosti offloadovat všechna mitigační pravidla.

si pak v software z každého přijatého paketu může vyčíst značku a podle ní pak předat pakety ke zpracování jednotlivým navazujícím mitigačním modulům.

V ideálním případě by bylo možné provést přímou transformaci klasifikačních pravidel existujícího klasifikátoru (popřípadě RTE ACL klasifikátoru) do tvaru RTE Flow pravidel. Tato transformace by měla dle dokumentace [3] být proveditelná. Při ověřování funkcionality na dostupných kartách *NVIDIA ConnectX-6 Dx 100GbE* a *Intel E810-CQDA2* bylo však zjištěno, že mnoho dokumentovaných funkcí není na těchto kartách podporováno. Toto testování probíhalo pomocí nástroje *testpmd* (sekce 3.6), který je distribuován jako součást frameworku DPDK a umožňuje ovládat síťovou kartu z příkazového řádku. Výstupy experimentů jsou uvedeny v příloze A. Je zřejmé, že síťová karta *Intel E810* (sekce A.1) nepodporuje celou řadu klíčových funkcí RTE Flow, mezi něž patří například výchozí klasifikační pravidla, možnost použití více úrovní priorit nebo více skupin pravidel. Síťová karta *NVIDIA ConnectX-6* (sekce A.2) sice také nepodporuje celou škálu RTE Flow funkcí (omezení na 4 prioritní úrovně, chybějící rozsahy portů), pro akceleraci aplikace je ale zřejmě výhodnější a dále uvedený návrh je tak zaměřen na tuto kartu.

Výběr offloadovaných pravidel

Dalším problémem, který brání přímému převodu pravidel z formátu RTE ACL do formátu RTE Flow je přítomnost pouze 4 možných hodnot priorit. Toto dokazuje výpis A.5, kde je vidět zamítnutí vložení pravidla s prioritou 4 (indexováno od nuly). Toto omezení bohužel není možné obejít s použitím sady pravidel, která je použita v rámci RTE ACL klasifikátoru. Je tedy nutné z celé sady vybírat pouze určitá pravidla, která se dají považovat za vhodná.

Pokud však budou existovat pravidla, která není možné z nějakého důvodu na síťovou kartu nahrát, je třeba udržovat navíc ještě záložní klasifikátor, například výše uvedený RTE ACL klasifikátor. Takovou situaci zachycuje schéma na obrázku 5.5. Vidíme, že do síťové karty se offloaduje jen podmnožina pravidel RTE ACL klasifikátoru. Při klasifikaci je pak část paketů označena pomocí akce *MARK* a část zůstává neoznačena. V rámci mitigačního filtru jsou pak neoznačené pakety dodatečně klasifikovány záložním klasifikátorem.

Kromě těchto omezení je také nutné brát v potaz omezenou kapacitu pravidel v síťové kartě. V rámci této práce nebyla sice kapacita pravidel omezující, pro budoucí rozšíření by však bylo vhodné tuto kapacitu ověřit.

Samotný výběr offloadovaných pravidel potom může být realizován následujícími způsoby:

Podle priorit Pravděpodobně nejsnazší možností, jak vybírat pravidla k offloadu na síťovou kartu je použít sadu pravidel RTE ACL klasifikátoru a vybrat z ní pravidla s nejvyšší prioritou. Pokud je tak uživatelem specifikována sada pravidel bez překryvů, pak jsou jednoduše nahrána všechna pravidla. Jinak jsou nahrána pseudopravidla, která ale pokrývají největší množství původních pravidel. Toto řešení je snadné a může být relativně efektivní pro rozumné sady pravidel. Bohužel, pokud mezi pravidly existuje mnoho překryvů, nemusejí offloadovaná pravidla odpovídat těm nejvíce využívaným a nárůst výkonu pak nebude tak velký.

Podle využití Další možností, jak vybírat pravidla pro přenos na síťovou kartu je zahájit klasifikaci pouze s využitím záložního klasifikátoru a ukládat informace o nejvíce využívaných pravidlech. U této podmnožiny pravidel je pak větší šance, že mezi pravidly neexistuje překryv a je možné je přímo nahrát do hardware. Toto řešení může využívat statistik, které jsou udržovány jednotlivými mitigačními metodami a je možné zjistit, která pravidla klasifikovala nejvíce paketů v uplynulém mitigačním okně. Offloadováním pouze nejvíce využívaných pravidel je možné zvýšit efektivitu hardwarové akcelerace, nicméně se zvyšuje režie při výměně mitigačních oken.

Spekulativně s využitím AGE akce Jednou z akcí, které jsou poskytnuty v rámci RTE Flow je akce AGE. Pomocí této akce lze realizovat spekulativní offloadování mitigačních pravidel. To může fungovat tak, že nejprve je do síťové karty nahrána libovolná podmnožina nepřekrývajících se pravidel. Na způsobu výběru této podmnožiny v tomto případě nezáleží, protože každé pravidlo má nastavenu akci *age* s časovačem nastaveným na délku mitigačního okna. Při každé změně mitigačního okna jsou pak pravidla, kterým vypršel časovač odstraněna a je možné je nahradit dalšími pravidly. Tímto způsobem je zajištěno, že v síťové kartě jsou po určitém čase přítomna jen ta pravidla, která jsou využívána.

Modifikace nahrané sady pravidel

Další situací, které je třeba řešit odlišným způsobem, než v případě softwarového klasifikátoru, je způsob modifikace aktuálně offloadované sady pravidel. Zatímco u RTE ACL klasifikátoru je možné změnu sady pravidel provést atomicky pouhou výměnou ukazatelů na používané datové struktury, RTE Flow využívá pro všechna pravidla společnou paměť. RTE Flow neposkytuje žádnou možnost atomické aktualizace klasifikačních pravidel, a je tedy nutné navrhnout odlišný způsob aktualizace sady pravidel. Aktualizace může probíhat několika zbůsoby:

Kompletní nahrazení sady pravidel K aktualizaci pravidel je možné využít záložního softwarového klasifikátoru a při potřebě aktualizovat pravidla neprve všechna RTE Flow pravidla odstranit, pak provést aktualizaci softwarového klasifikátoru (předpokládáme, že ta může proběhnout atomicky) a nakonec nahrát nová RTE Flow pravidla. Tímto způsobem je zajištěno, že mitigační filtr bude vždy v konzistentním stavu a nebude docházet k chybné klasifikaci.

Zásadní nevýhodou tohoto přístupu je ale pokles výkonu aplikace při aktualizaci pravidel, protože po určitý čas bude velkou část paketů klasifikovat pouze softwarový

klasifikátor. Doba nahrávání a odstraňování pravidel navíc není nijak specifikovaná a je závislá na konkrétní síťové kartě.

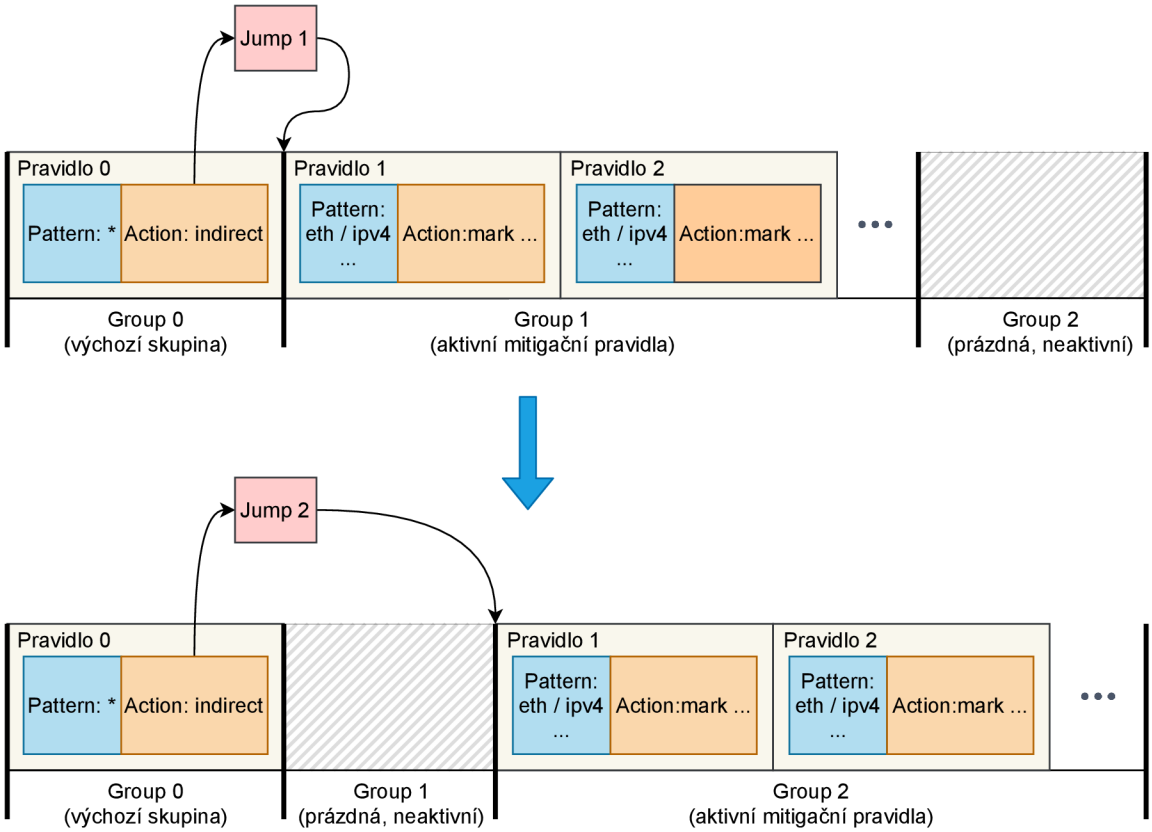
Nahrazení změněných pravidel Pro eliminaci nutnosti všechna pravidla mazat a znovu nahrávat je možné zjistit rozdíl mezi původní a novou sadou pravidel a aktualizovat pravidla postupně na základě tohoto rozdílu. I k tomuto způsobu aktualizace pravidel je třeba uchovávat záložní klasifikátor, který se postará o klasifikaci v době, kdy jsou některá pravidla odstraňována a jiná nahrávána. Toto řešení je v nejhorším případě stejně efektivní jako výše uvedená metoda aktualizace všech pravidel. Komplikací tohoto přístupu je nutnost implementovat mechanismus zjišťující rozdíl mezi sadami pravidel.

Použití více skupin Dalším možným řešením je využít atributu *group* (viz sekce 3.4), který specifikuje použitou skupinu pravidel a pro mitigační filtr použít dvě tyto skupiny. Toto řešení je naznačeno na obrázku 5.6. Vidíme, že jedna skupina obsahuje aktuálně používaná pravidla, druhá zůstává v běžném stavu prázdná. Při požadavku na aktualizaci pravidel budou tak nejprve nahrána pravidla do aktuálně prázdné skupiny, následně bude provedena výměna aktivní skupiny a nakonec se z původně aktivní, nyní již deaktivované skupiny odstraní všechna pravidla. Tímto způsobem je možné zajistit aktualizaci pravidel bez nutnosti uchovávat záložní klasifikátor. K výměně používaných skupin pak může být použito dvou speciálních akcí poskytovaných v rámci RTE Flow: INDIRECT a JUMP. Akce indirect slouží pro sdílení stejné akce mezi více pravidly. Ke konfiguraci indirect akcí je rovněž poskytováno speciální rozhraní. Díky tomu je indirect akce jedinou akcí, kterou je možné aktualizovat *in place*, bez nutnosti jejího odstranění a opětovného nahrání. Akce jump pak dovoluje přesměrovat pakety do jiné skupiny pravidel v síťové kartě.

Celkem je tak specifikováno pravidlo v rámci výchozí skupiny (skupina 0), které klasifikuje všechny pakety a jehož akcí je indirect akce odkazující se na akci jump. Ta pak přesměruje pakety do jedné ze dvou skupin vyhrazených pro mitigační pravidla. Využití akce typu indirect v popsaném případě není zcela nutné, protože v rámci výchozí skupiny je jen jedno pravidlo. V případě, že by ale v rámci výchozí skupiny bylo obsaženo více pravidel, je pak indirect akce jedinou možností, jak sdílet akce mezi pravidly.

Použití více skupin a priorit Indirect akce jsou na použité síťové kartě *NVIDIA ConnectX-6* podporované, bohužel ale ovladač této karty zakazuje vnoření akce JUMP, což je zobrazeno ve výpisu A.6. Je tedy nutné najít jiný způsob, jak atomicky aktualizovat použitou skupinu. K tomuto je možné využít omezeného počtu priorit a vytvářet více pravidel. Popsaný způsob aktualizace pravidel v jednotlivých fázích znázorňuje obrázek 5.7 a způsob realizace nástrojem *testpmd* je naznačen ve výpisu A.7. Jednotlivé fáze na obrázku probíhají následovně:

1. Existuje pravidlo s výchozím klasifikačním vzorem a akcí určující skupinu 1.
2. Vytvoří se nové pravidlo se stejným klasifikačním vzorem, akcí určující skupinu 2, ale s prioritou zvýšenou o 1. Tímto je původní pravidlo zastíněno novým s vyšší prioritou.
3. Původní pravidlo je odstraněno a zůstává jen pravidlo s vyšší prioritou (skok do skupiny 2).



Obrázek 5.6: Schéma aktualizace RTE Flow pravidel s využitím skupin a akce *indirect*.

4. Vloženo je pravidlo se stejným vzorem a akcí (skok do skupiny 2) jako pravidlo s vyšší prioritou, ale tentokrát s prioritou původního pravidla.
5. Pravidlo s vyšší prioritou je odstraněno.

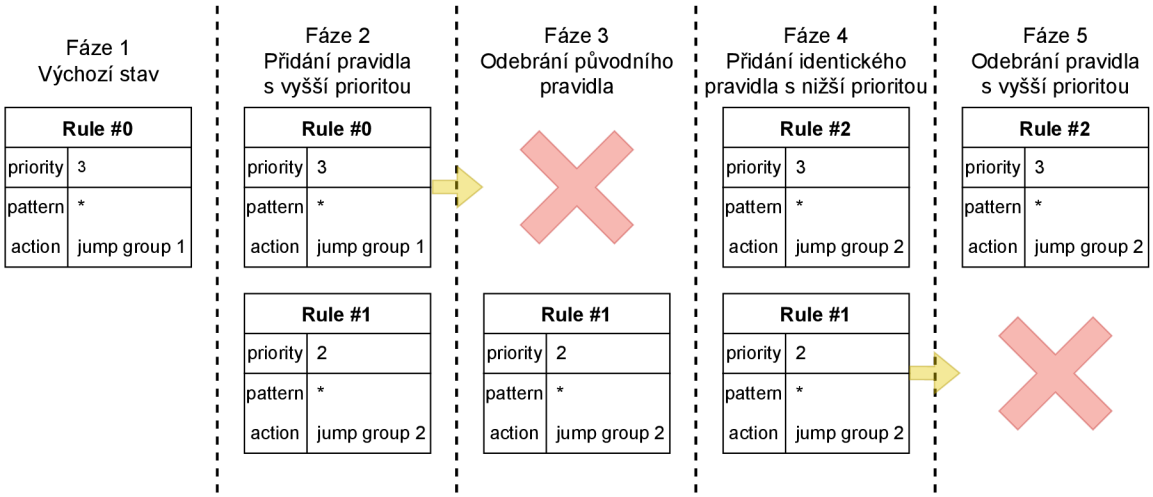
5.4 Akcelerace dalších komponent

Jak bylo zmíněno v sekci 4.2, klasifikace paketů je časově nejnáročnější část zpracování provozu aplikací DDoS Protector. Kromě akcelerace této komponenty je ale možné aplikaci akcelerovat i jinými způsoby. Tyto návrhy nejsou v rámci této práce implementovány, slouží však jako náměty k budoucímu rozšíření.

Optimalizace komponenty Packet Router

Existující komponenta *Packet Router* (viz kapitola 4), používá ke své činnosti dvě instance LPM modulů, které odděleně klasifikují IPv4 a IPv6 pakety. Nahrazení těchto dvou komponent za jednu by mohlo přinést další zvýšení výkonu díky absenci rozdělování paketů mezi IPv4 a IPv6 a následné sjednocování obou skupin.

Jednou z potenciálních DPDK knihoven vhodných k této úpravě je opět RTE ACL, která poskytuje jednotné rozhraní pro oba typy paketů. Klasifikace typu *longest prefix match* je možné v rámci ACL docílit díky nastavení vyšší priority u pravidel s vyšší délkou prefixu.



Obrázek 5.7: Schéma pěti fází atomické modifikace akce *jump* v RTE Flow.

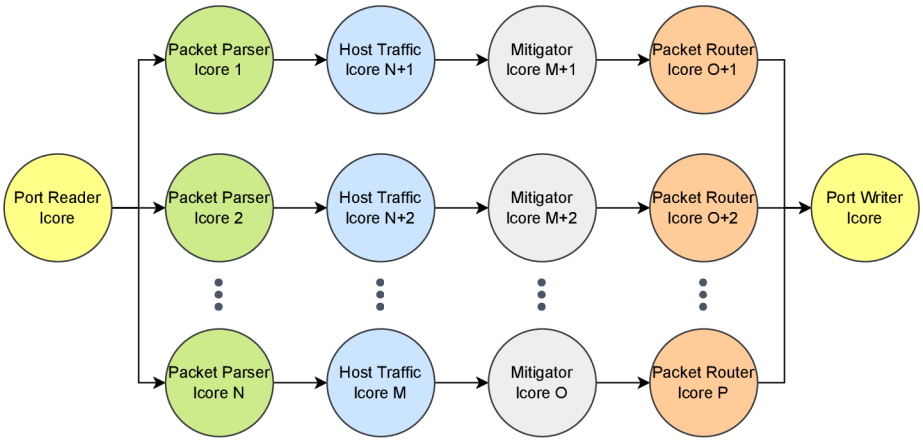
Komponentu *Packet Router* je také možné akcelerovat s využitím RTE Flow. Pravidla RTE Flow pak budou klasifikovat na základě hlaviček IP a podle adresy cílové sítě bude využita akce *MODIFY_FIELD* pro aktualizaci MAC adresy paketu a dekrementaci čítače *Time To Live*.

Úprava způsobu paralelizace

Sekce 3.3 zmiňuje dva způsoby paralelizace výpočtu DPDK aplikací, avšak DDoS Protector v současné době využívá pouze model *run-to-completion*. Jednou z možností akcelerace aplikace na vícejádrových systémech je spojení používaného modelu *run-to-completion* s *pipelined* modelem, což je zobrazeno na obrázku 5.8. Vidíme, že každá komponenta může individuálně pracovat v režimu *run-to-completion* a mezi komponentami je použit *pipelined* model.

Jednotlivá jádra tak vždy vykonávají jen část celkové zřetězené linky. To je mimo jiné výhodné i z pohledu přístupu do paměti, protože každé jádro pracuje nad vlastními datovými strukturami, které mohou umístit do cache blízko danému jádru.

Taková optimalizace však vyžaduje úpravu všech existujících komponent aplikace. Bylo by totiž nutné umožnit komunikaci jednotlivých komponent mezi sebou pomocí struktury *rte_ring*, což s sebou nese určitou míru režie.



Obrázek 5.8: Kombinace modelu *run-to-completion* a *pipelined* modelu v rámci architektury aplikace DDoS Protector.

Kapitola 6

Implementace

V rámci této práce vznikly nové typy paketových klasifikátorů. Pro možnost vybrat typ používaného klasifikátoru byla vytvořena továrna paketových klasifikátorů, která je detailněji popsaná v sekci 6.1.

V souvislosti s nutností předzpracování sady pravidel pro jejich instalaci (viz návrh v sekci 5.1) byla implementována komponenta *non_ov_ruleset*, která sestavuje sadu pravidel s odstraněnými seznamy sítí a rozsahů portů. Tato komponenta je detailněji popsána v sekci 6.2.

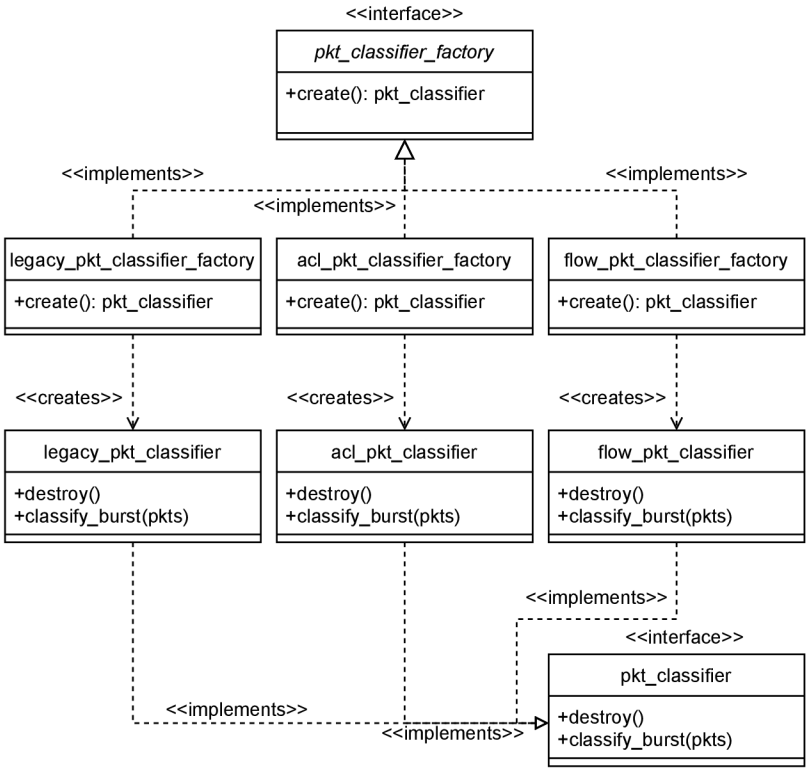
Pro zpětný převod výsledků klasifikace do podoby pravidel mitigačního jádra byla implementována komponenta *pseudorule_table*, která je detailněji popsána v sekci 6.3.

S pomocí uvedených komponent bylo nakonec možné implementovat dva typy klasifikačních komponent. Klasifikátor s využitím knihovny RTE ACL je uvedený v sekci 6.5, hardwarově akcelerovaný RTE Flow klasifikátor pak v sekci 6.6. Integrace obou klasifikačních komponent v rámci aplikace DDoS Protector je popsána v sekci 6.7.

6.1 Továrna paketových klasifikátorů

Jedním z kroků implementace je zavést mechanismus, který umožní vytvářet paketové klasifikátory daného typu. Vhodným řešením z pohledu objektově orientovaného návrhu je využít přístup zobrazený v diagramu tříd na obrázku 6.1. V tomto diagramu je vidět použití návrhového vzoru *abstraktní továrna*. Existuje tak jedno bázové rozhraní (*pkt_classifier_factory*), které nemá implementaci (v diagramu zobrazeno nahoře uprostřed). Toto rozhraní pak implementují konkrétní továrny pro jednotlivé typy paketových klasifikátorů. Továrny poskytují jedinou metodu *create*, kterou vznikají instance klasifikátoru. Ty jsou předávány mitigačnímu jádru opět jako abstraktní typ *pkt_classifier* a každá instance poskytuje metody *classify_burst* pro klasifikaci paketů a *destroy* pro dealokaci použitých struktur.

Díky využití tohoto návrhového vzoru je možné vytvářet nové typy klasifikátorů na základě stejného rozhraní a v aplikaci je pak libovolně zaměňovat, je tedy dosaženo dynamického polymorfismu. Díky tomuto bylo možné dále implementovat jak klasifikátor s využitím DPDK knihovny RTE ACL, tak i klasifikátor s využitím knihovny RTE Flow.



Obrázek 6.1: Diagram tříd pro modul vytváření paketových klasifikátorů s využitím abstraktní továrny.

6.2 Nepřekrývající se sada pravidel

Jedním z výše uvedených problémů implementace paketového klasifikátoru s využitím knihovny RTE ACL je nutnost provést expanzi pravidel na *pseudopravidla*. K tomuto účelu byl implementován modul *non_ov_ruleset* (*non-overlapping ruleset*, nepřekrývající se sada pravidel).

Odstranění seznamů sítí a seznamů rozsahů portů

Prvním krokem převodu původní sady pravidel je odstranění seznamů IP prefixů a seznamů portů. K tomuto byl implementován následující naivní algoritmus realizující kartézský součin mezi všemi seznamy popsaný ve výpisu 1. Vstupem algoritmu je pravidlo, které obsahuje množiny zdrojových a cílových IP adres a množiny rozsahů zdrojových a cílových portů. Navíc je součástí pravidla vždy jedna hodnota identifikátoru VLAN. Algoritmus na vstupu vyžaduje alespoň jednu cílovou síť a jeden cílový rozsah portů transportní vrstvy, což je implementační podmínka vycházející z požadavků mitigačního jádra. V případě, že je množina cílových sítí nebo cílových rozsahů portů prázdná, algoritmus končí s chybou (řádky 1, 2). Protože pro zdrojové sítě a rozsahy portů tato podmínka není, je nutné v dalším kroku zkontrolovat jejich prázdnost a popřípadě je inicializovat hodnotou reprezentující celý prostor IP adres a portů transportní vrstvy (řádky 4 – 10). Následuje výpočet samotného kartézského součinu mezi všemi vstupními množinami, z nichž vzniká několik nových pra-

Algoritmus 1: Eliminace seznamů IP prefixů a rozsahů pro jedno pravidlo

Vstup : Původní pravidlo $(vlan, srcIps, dstIps, srcPorts, dstPorts)$.

```
1  if dstIps = ∅ ∨ dstPorts = ∅ then
2  |  Skonči s chybou. /* Alespoň jeden cílový prefix a jeden cílový rozsah
   |     musí být vždy zadán                                              */
3  end
4  Inicializuj prázdný seznam list.
5  if srcIps = ∅ then
6  |  let srcIps = {0.0.0.0/0}
7  end
8  if srcPorts = ∅ then
9  |  let srcPorts = {0 − 65535}
10 end
11 for srcIp ∈srcIps do
12 |  for dstIp ∈dstIps do
13 |  |  for srcPort ∈srcPorts do
14 |  |  |  for dstPort ∈dstPorts do
15 |  |  |  |  let newRule = (vlan, srcIp, dstIp, srcPort, dstPort)
16 |  |  |  |  Přidej newRule do list.
17 |  |  |  end
18 |  |  end
19 |  end
20 end
```

Výstup: Seznam upravených pravidel s odstraněnými seznamy.

videl, které jsou postupně vkládány do výstupního seznamu pravidel (řádky 11 – 20). Tento algoritmus vykazuje velmi špatnou časovou složitost, nicméně pro tento účel je dostačující.

Odstranění překryvů mezi pravidly

Dalším krokem převodu je detekce překryvů v nově vzniklém seznamu pravidel s odstraněnými seznamy. Tento problém je možné definovat jako hledání průniků množin ve více dimenzích – odděleně pro zdrojový IP prefix, cílový IP prefix, zdrojový rozsah portů a cílový rozsah portů. Převedeme-li navíc IP prefixy na rozsahy hodnot, pak se ve všech případech jedná o uspořádané množiny definované svojí spodní a horní hranicí, což dále zjednodušuje detekci.

Můžeme definovat množiny $A = \langle a_{low}, a_{high} \rangle$ a $B = \langle b_{low}, b_{high} \rangle$, které reprezentují rozsahy IP adres nebo portů. Pak pro tyto množiny platí:

$$b_{low} \leq a_{low} \wedge b_{high} \geq a_{high} \implies A \subseteq B$$
$$a_{high} \geq b_{low} \wedge a_{low} \leq b_{high} \implies A \cap B \neq \emptyset$$

Pro detekci překryvů mezi dvěma pravidly je pak možné vycházet z kombinace průniků mezi jednotlivými součástmi s výjimkou hodnoty VLAN identifikátoru, který je vždy zadán přesně a je porovnáván přímo. Mějme tedy pravidla $P_1 = (vlan_1, sIp_1, dIp_1, sPort_1, dPort_1)$ a $P_2 = (vlan_2, sIp_2, dIp_2, sPort_2, dPort_2)$, pak mohou nastat tři situace:

1. Pravidlo P_1 je plně překryto pravidlem P_2, tedy:

$$vlan_1 = vlan_2 \wedge sIp_1 \subseteq sIp2 \wedge dIp_1 \subseteq dIp_2 \wedge sPort_1 \subseteq sPort_2 \wedge dPort_1 \subseteq dPort_2 \implies P_1 \subseteq P_2$$

2. Pravidlo P_1 není plně překryto pravidlem P_2, nicméně existuje mezi nimi průnik:

$$(vlan_1 = vlan_2) \wedge (sIp_1 \cap sIp2 \vee dIp_1 \cap dIp_2 \vee sPort_1 \cap sPort_2 \wedge dPort_1 \cap dPort_2) \implies P_1 \cap P_2 \neq \emptyset$$

3. Mezi pravidly neexistuje žádný průnik:

$$P_1 \cap P_2 = \emptyset$$

Následně je možné aplikovat algoritmus pro expanzi pravidel uvedený ve výpisu 2. Jeho vstupem je sada pravidel získaná aplikací předchozího algoritmu 1. Všem pravidlům v této sadě je nejprve přiřazena nejnižší možná priorita a následuje cyklus s předem nedefinovaným počtem iterací (řádek 3). Uvnitř tohoto cyklu je zanořen další cyklus, který vybírá každou dvojici pravidel v seznamu a z průniku této dvojice pravidel sestavuje nové pseudopravidlo k (řádky 5, 6). V případě, že k bylo možné sestavit (mezi pravidly existoval průnik) a k dosud nenáleží do sady pravidel, je pseudopravidlu zvýšena priorita a je vloženo do existující sady pravidel (řádky 7, 8). Protože ale tímto vložením vznikla nová sada, kde pseudopravidla mohou vznikat i průnikem s pseudopravidlem k, je nutné vnitřní cyklus znovu opakovat (řádek 9). Celkově se tento cyklus opakuje do té doby, než jsou všechny možné průniky pravidel již zahrnuty v sadě pravidel, kdy je možné výpočet ukončit (řádek 12). Výstupem tohoto algoritmu je sada pravidel, která má překryvy pravidel vyřešeny přidáním pseudopravidel s vyšší prioritou. Takovou sadu pravidel je pak jednoduše možné převést do formátu, který je očekáván knihovnami RTE ACL nebo RTE Flow.

Uvedený algoritmus má opět špatnou časovou složitost, protože se výpočet opakuje znovu při přidání nového pseudopravidla. Jako jistá optimalizace bylo implementováno prvotní rozdělení sady pravidel do několika seznamů podle hodnot VLAN identifikátorů, mezi kterými nemohou existovat průniky, čímž se sníží počet prvků, pro které je nutné detekci průniků provádět. Toto rozdělení může být dále rozšířeno i na jiné položky, než jen VLAN identifikátor.

6.3 Tabulka pseudopravidel

Na základě algoritmů uvedených v sekci 6.2 je možné vytvořit sadu pravidel identifikovanými překryvy a tuto sadu pak použít při inicializace klasifikátoru s využitím RTE ACL knihovny. Kromě toho je však potřeba zajistit i zpětný převod výsledků klasifikace a aktualizaci datových struktur původních mitigačních pravidel v mitigačním jádře.

Za tímto účelem byl vytvořen modul *pseudorule_table*, který zajišťuje vyhledání identifikátorů původních mitigačních pravidel na základě výsledků klasifikace modulem RTE ACL.

Schéma implementace této tabulky je na obrázku 6.2a. Jednou z žádaných vlastností této komponenty je velmi rychlé vyhledání hodnot pro jakýkoliv klíč. Z tohoto důvodu bylo hlavní datovou strukturou zvoleno obyčejné pole (na obrázku žlutou barvou), které bude přímo adresované výsledkem klasifikačního algoritmu. Pro uložení identifikátorů původních

Algoritmus 2: Detekce překryvů mezi dvojicemi pravidel a vytvoření pseudopravidel

Vstup : Sada pravidel s odstraněnými seznamy S

```
1  let finish = False
2  Každému pravidlu je přiřazena implicitní priorita 0.
3  while finish = False do
4  |   label start
5  |   for (i, j) ∈ S do
6  |   |   let k = i ∩ j
7  |   |   if k ≠ ∅ ∧ k ∉ S then
8  |   |   |   Přidej k do S s prioritou zvýšenou o 1.
9  |   |   |   goto start
10 |   |   end
11 |   end
12 |   finish = True /* Všechna pseudopravidla už byla vytvořena a je
   |      možné cyklus ukončit.                                      */
13 end
```

Výstup: Sada pravidel, kde jsou překryvy nahrazeny pseudopravidly s vyšší prioritou.

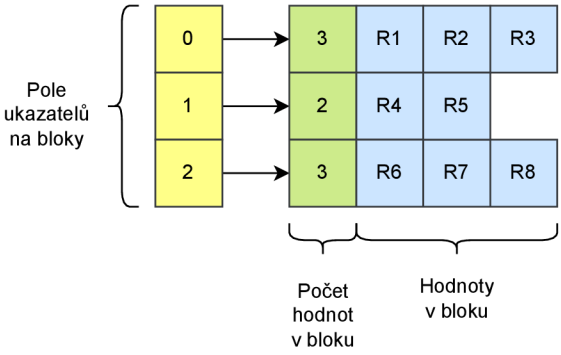
mitigačních pravidel náležících jednotlivým pseudopravidlům je pole rozděleno do bloků. Bloky jsou pak uvozeny číslem určujícím počet uložených záznamů (na obrázku zelenou barvou) a za tímto číslem pak následují samotné záznamy (na obrázku modrou barvou).

Při vytváření tabulky jsou bloky přidávány postupně na konec tabulky na základě poskytnuté sady pravidel. Funkce přidávající bloky do tabulky pak na svém výstupu vrací index uloženého bloku, který následně slouží jako identifikátor pseudopravidla v ACL klasifikačním algoritmu. Vyhledání bloku v tabulce je pak proveden jednoduchým přístupem na blok odpovídající výsledku klasifikace.

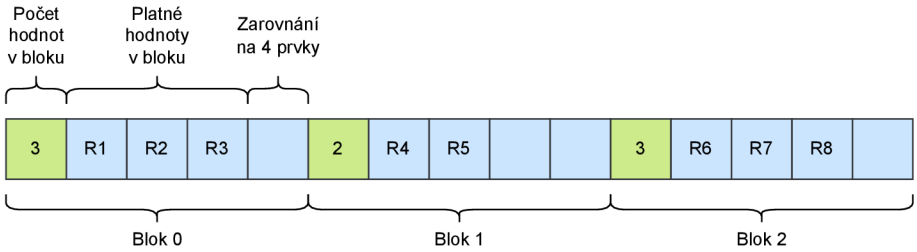
Postupná alokace nových bloků však způsobuje, že jednotlivé bloky mohou být v paměti programu umístěny zcela nahodile, což ve spojení s nezarovnanou alokací zamezuje efektivnímu přístupu k blokům pomocí paměti cache a přednačítání (*prefetching*) dat z paměti. Pro zvýšení výkonnosti je možné tuto vyhledávací strukturu optimalizovat na základě dvou hlavních požadavků:

1. Zarovnání jednotlivých bloků na velikost, která je soudělná s velikostí řádku cache.
2. Využití spojitého úseku paměti.

Schéma realizace takto optimalizované varianty tabulky pseudopravidel je zobrazeno na obrázku 6.2b. Vidíme, že bloky v paměti jsou uloženy spojitě za sebou a stejně jako v předchozí variantě jsou bloky uvozeny počtem skutečně uložených hodnot (na obrázku zelenou barvou) za nímž následují samotné bloky (modrá barva). Zarovnání bloků v paměti při zachování možnosti uložení libovolného počtu hodnot bylo dosaženo alokací bloků jako *Variable Length Array* (pole proměnné délky, *VLA*) a vynucení jejích zarovnání pomocí atributu *aligned* při deklaraci struktury bloku. Z pohledu jazyka C má pak jeden blok velikost (získanou operátorem *sizeof*) danou hodnotou atributu *aligned* a pomocí této hodnoty umožňuje indexování v tabulce, avšak skutečná velikost bloku je určená až při jeho alokaci při přidávání bloků do tabulky. Toto umožňuje zachovat přístup do tabulky zalo-



(a) Jednoduchá implementace.



(b) Optimalizovaná implementace.

Obrázek 6.2: Srovnání jednoduché a optimalizované implementace tabulky pseudopravidel.

žený na přímé indexaci konkrétního bloku, nicméně je třeba brát v potaz, že při indexaci do špatného bloku jsou získána data patřící cizímu bloku nebo data naprosto nesmyslná.

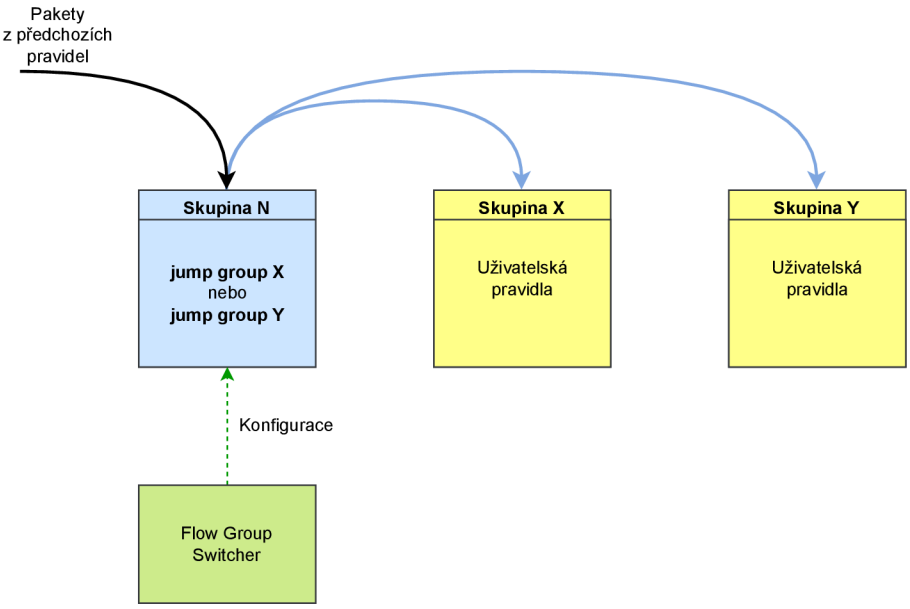
6.4 Komponenta pro změnu RTE Flow skupin

V souvislosti s implementací klasifikátoru využívajícího offloadu klasifikačních pravidel do hardware pomocí knihovny RTE Flow bylo třeba řešit problém aktualizace sady pravidel instalované do síťové karty. K tomuto účelu je možné využít několika přístupů, jež jsou popsané v sekci 5.3. Jelikož podpora offloadu pomocí rozhraní RTE Flow se v závislosti na typu použité síťové karty liší, je možné použít různé přístupy.

V rámci této práce bylo definováno rozhraní *Flow Group Switcher*, jehož cílem je vytvořit abstrakci pro výměnu aktuálně používané skupiny pravidel různými způsoby v závislosti na použité síťové kartě. Toto rozhraní poskytuje 3 metody:

1. Zjištění aktivní skupiny pravidel.
2. Zjištění neaktivní skupiny pravidel.
3. Výměna aktivní skupiny pravidel za neaktivní.

Společně s definicí rozhraní *Flow Group Switcher* byla implementována i konkrétní komponenta specifická pro síťové karty *NVIDIA ConnectX-6* využívající principu zobrazeného na schématu na obrázku 5.7. Schéma použití této komponenty je zobrazeno na obrázku 6.3. *Flow Group Switcher* (na obrázku znázorněn zeleně) má v základní podobě na starosti právě jedno RTE Flow pravidlo (na obrázku modrou barvou), které je umístěno do své vlastní skupiny. Toto pravidlo realizuje akci *jump* s cílem v jedné ze dvou připojených



Obrázek 6.3: Schéma použití komponent Flow Group Switcher pro efektivní výměnu aktivních skupin pravidel.

skupin (na obrázku znázorněny žlutě) a obsahuje pouze prázdný *mitigation pattern*, tedy klasifikuje všechny pakety.

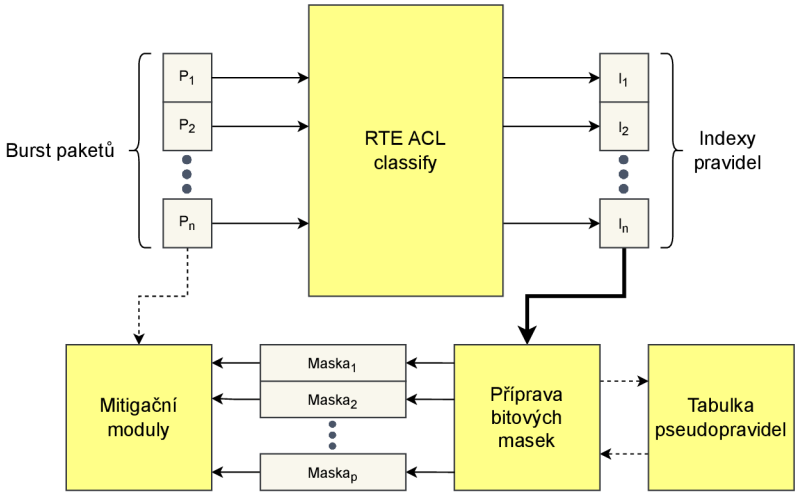
Jako rozšíření této komponenty byla přidána podpora pro volitelnou inicializaci výchozího pravidla v obou připojených skupinách.

6.5 RTE ACL paketový klasifikátor

Jednou z hlavních součástí této práce byla implementace dvou typů paketového klasifikátoru. Prvním z nich je klasifikátor využívající DPDK knihovnu RTE ACL. Implementace této komponenty využívá komponentu pro expanzi pseudopravidel uvedenou v sekci 6.2 a tabulku pro jejich uložení a rychlé vyhledání popsanou v sekci 6.3.

Nová instance klasifikátoru je vytvářena vždy při detekci změny mitigačních pravidel. Důležitou vlastností je uložení této nové instance do aktuálně nepoužívaného mitigačního okna. V okamžiku výměny mitigačního okna je pak nová instance aktivována, což vede k atomickému nahrazení klasifikačních pravidel.

Samotná klasifikace je pak zobrazena ve schématu na obrázku 6.4. Nejprve je volána klasifikační funkce z rozhraní RTE ACL knihovny, která na svém vstupu přijímá burst paketů a jejím výstupem je pole celočíselných hodnot, které reprezentují indexy pravidel klasifikující konkrétní pakety. Následně je nutné výsledky klasifikace promítnout do původních mitigačních pravidel, čehož je možné dosáhnout přípravou bitových masek pro každé původní pravidlo, jak je uvedeno v sekci 4.4. V rámci získaných výsledků klasifikace se však nacházejí jak indexy původních pravidel, tak i pseudopravidel a je potřeba zpětně získat indexy původních pravidel. K tomuto je využita dříve implementovaná tabulka pseudopravidel (sekce 6.3), která efektivně vyhledá blok odpovídající danému pravidlu a ten předá na výstup. V cyklu jsou tak procházeny všechny klasifikované pakety a postupně jsou na-



Obrázek 6.4: Schéma klasifikace paketů s využitím klasifikátoru RTE ACL.

staveny bitové masky odpovídající indexům pravidel získaných z tabulky. Připravené bitové masky jsou pak společně s původním burstem paketů předány jednotlivým aktivním mitigačním modulům, které provedou výsledné zpracování paketů.

Jedním z problémů, které byly při implementaci objeveny, byl *race condition* při přístupu k bitovým maskám jednotlivých pravidel, které byly implementovány jako sdílená proměnná. V případě klasifikátoru však není nutné bitové masky sdílet dohromady mezi více vlákny, a proto mohla být implementace upravena tak, že si každé vlákno ukládá vlastní kopii bitových masek, se kterou pracuje. Toto řešení je vhodné i z toho důvodu, že jednotlivé mitigační moduly navazující na paketový klasifikátor umožňují pracovat s daty na úrovni jednotlivých vláken. Zmíněná úprava tedy znamenala přidání dvou metod paketového klasifikátoru určených pro alokaci a uvolnění paměti pro datové struktury lokální pro dané vlákno.

6.6 RTE Flow paketový klasifikátor

Dalším krokem byla implementace paketového klasifikátoru akcelerovaného offloadem části pravidel do hardware pomocí rozhraní RTE Flow. Tato implementace staví na návrhu uvedeném v sekci 5.2. Implementovaná varianta offloaduje do hardware jen podmnožinu pravidel s nejvyšší prioritou, která nemá překryvy mezi pravidly, avšak teoreticky by ylo možné offloadovat až tři nejvyšší úrovně (čtvrtá úroveň je vyhrazena pro výchozí pravidlo). Pro uložení zbylých pravidel využívá záložního RTE ACL klasifikátoru.

Převod rozsahů na prefixy

Prvním objeveným problémem byla nemožnost zadávat rozsahy portů. RTE Flow sice ve specifikaci pravidla umožňuje nastavit hodnotu *last*, díky níž je možné rozsahy definovat, ovladač síťové karty *NVIDIA ConnectX-6* však tuto možnost pro TCP a UDP porty nepodporuje. Výpis A.4 ukazuje pokus o vytvoření pravidla s rozsahem TCP portů. Je vidět, že chybová hláška pochází přímo od PMD ovladače použité síťové karty.

Řešením pro rozsahy portů je jejich převod do podoby několika prefixů. K tomu je možné využit rekurzivního algoritmu 3. Ten nejprve zkontroluje, zda aktuálně nastavené meze, které jsou vždy mocninami dvojky, náleží do zpracovávaného rozsahu $\langle low, high \rangle$, (řádek 2). Tato podmínka je zároveň i podmínkou rekurze, při jejímž splnění je aktuálně nastavený interval $\langle x_1, x_2 \rangle$ triviálně převeden na prefix a vložen do výstupní množiny prefixů (řádek 3). Pokud podmínka není splněna, pokračuje se další podmínkou, která hlídá, zda hledaný rozsah alespoň částečně náleží do aktuálně nastaveného intervalu $\langle x_1, x_2 \rangle$ (řádek 5) a následně je provedeno binární půlení tohoto intervalu (řádky 6, 7) a algoritmus je volán rekurzivně pro každou polovinu intervalu (řádky 8, 9). Výstupem algoritmu je pak množina prefixů, která odpovídá původnímu rozsahu $\langle low, high \rangle$.

Uvedený algoritmus je převzat z [24]. Tato práce také obsahuje důkaz, že z rozsahu $\langle 0, 2^s \rangle$ může vzniknout až $2s - 2$ odlišných prefixů, což umožňuje algoritmus implementovat efektivně s použitím pole pevné velikosti, která může být v případě portů transportní vrstvy $2 \times 16 - 2 = 30$.

Algoritmus 3: Převod rozsahu na množinu prefixů

Vstup : Hranice rozsahu low, $high$, aktuálně zpracovávané hranice prefixu x_1, x_2.

1 První volání je provedeno s $x_1 = 0$ a $x_2 = 2^s - 1$, kde s je bitová šířka rozsahu.
2 **if** $\langle x_1, x_2 \rangle \in \langle low, high \rangle$ **then**
3 | Přidej prefix s rozsahem $\langle x_1, x_2 \rangle$ do výstupní množiny prefixů.
4 **else**
5 | **if** $x_1 \neq x_2 + 1 \wedge ((low \in \langle x_1, x_2 \rangle) \vee (high \in \langle x_1, x_2 \rangle))$ **then**
6 | | **let** $x_3 = x_1 + \frac{x_2 - x_1}{2}$
7 | | **let** $x_4 = x_3 + 1$
8 | | Rekurzivně zavolej algoritmus s argumenty $low, high, x_1, x_3$.
9 | | Rekurzivně zavolej algoritmus s argumenty $low, high, x_2, x_4$.
10 | **end**
11 **end**

Výstup: Množina prefixů odpovídajících rozsahu $low, high$.

Úprava hlavní smyčky mitigačního jádra

Největším problémem implementace akcelerovaného paketového klasifikátoru byla realizace výměny pravidel, která musela být provedena tak, aby nedošlo k chybné konzistenci sady pravidel mezi offloadovanými pravidly v síťové kartě a pravidly instalovanými v záložním ACL klasifikátoru.

K tomuto bylo třeba upravit hlavní smyčku mitigační komponenty, jejíž základní podoba byla uvedena v sekci 4.3 a která se stará o korektní sestavení sady pravidel, jejich nahrání do mitigačního okna a následnou výměnu neaktivního okna za aktivní. Z principu fungování mitigační komponenty je nutné zajistit, aby při jakémkoliv selhání v rámci této smyčky nenastávaly nedefinované stavy a aplikace dále fungovala. Protože ale RTE Flow paketový klasifikátor přistupuje k hardware síťové karty, je třeba implementovat dodatečnou kontrolu korektnosti aktuálně používané sady RTE Flow pravidel. Tohoto bylo dosaženo spojením dvou přístupů:

Identifikace použitého klasifikátoru První úpravou bylo přidání identifikace instance klasifikátoru do klasifikovaných paketů označených akcí *MARK*. Tato identifikace vy-

užívá jednoho bitu v celkové značce. To je dostačující, jelikož v dalším kroku bylo zajištěno, že maximálně dva klasifikátory mohou mít v jeden okamžik offloadovaná pravidla. Při klasifikaci pak dochází k ověření, že uložený bit odpovídá hodnotě přiřazené konkrétní instanci klasifikátoru. V případě chybné identifikace je pak výsledek klasifikace v hardware zahozen a je nutné provést opětovnou klasifikaci pomocí záložní ACL komponenty.

Rozšíření rozhraní klasifikátoru Dalším krokem bylo rozšíření rozhraní klasifikátoru o metody, které jsou později volány v rámci hlavní smyčky mitigátoru. Tyto metody byly implementovány jako volitelné a implementace klasifikačních komponent se bez nich obejde, nevyžaduje-li offload pravidel do hardware. Konkrétně se jednalo o přidání následujících 4 metod:

pre_apply Tato metoda se používá bezprostředně před výměnou mitigačního okna. Jejím cílem je zajistit výměnu RTE Flow skupin použitých pro klasifikaci. Algoritmus pro výměnu skupin pravidel implementuje komponenta *Flow group switcher* popsaná v sekci 6.4.

post_apply Tato metoda je určena jako komplementární k metodě *pre_apply* a jejím cílem je zkontrolovat úspěšnost výměny skupiny pravidel. V případě, že je touto metodou detekováno selhání, je nutné uvolnit všechna offloadovaná pravidla daného klasifikátoru (pokud nějaká jsou).

offloads_insert Zodpovědností této metody je provádět offload RTE Flow pravidel. K tomuto účelu je nejprve získána skupina, do které budou pravidla nahrána a následně je inicializován identifikátor klasifikátoru. Nakonec je proveden offload celé sady pravidel. Selhání při nahrávání pravidel do síťové karty musí být v tomto případě řešeno již v rámci této metody odstraněním všech již nahraných pravidel.

offloads_remove Cílem této metody je odstranit pravidla z již nepoužívaného klasifikátoru před jeho samotnou dealokací.

V rámci hlavní mitigační smyčky (sekce 4.3) pak dochází k volání nově přidaných metod klasifikátoru, což mění funkcionalitu jejích jednotlivých fází tak, že:

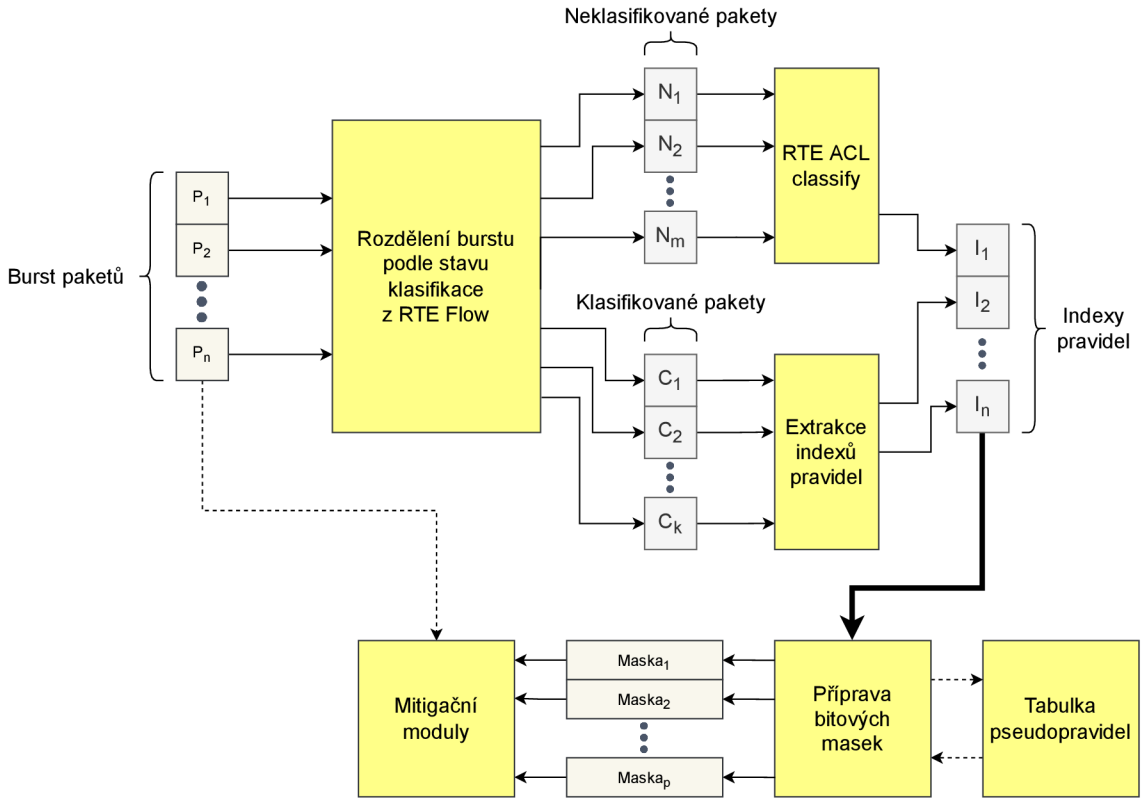
1. Fáze *swap* je rozšířena o volání metody *pre_apply* před samotnou výměnou mitigačních oken a metody *post_apply* po provedení této výměny. Na konci této fáze je navíc ještě volána metoda *offloads_remove* pro bezpečné odebrání offloadovaných pravidel uplynulého mitigačního okna.

2. Fáze *evaluate* je zachována beze změny.

3. Fáze *reload* je pak rozšířena o offloadování mitigačních pravidel příštího mitigačního okna v případě změny sady pravidel. K tomuto je volána metoda *offloads_insert*.

Úprava způsobu klasifikace paketů

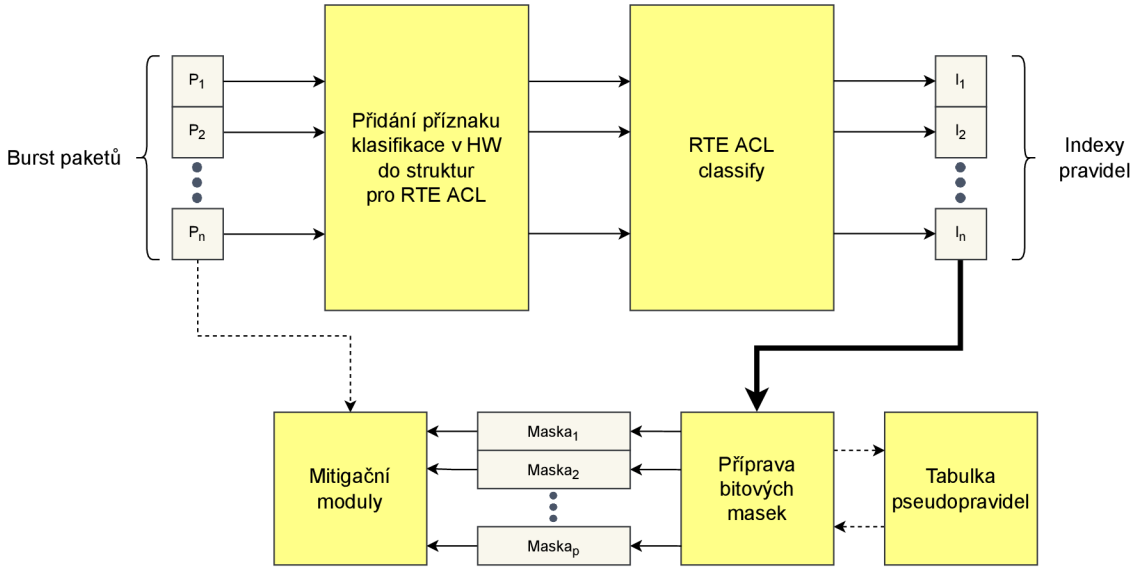
Z důvodů uvedených v podsekci 5.3 není možné offloadovat všechna mitigační pravidla, což znamená, že sada RTE Flow pravidel je podmnožinou sady pravidel instalované ACL komponentou. Při klasifikaci je tedy nutné nejprve z celého příchozího burstu paketů vybrat ty pakety, které dosud nebyly klasifikovány v hardware. Toho je možné dosáhnout dvěma způsoby, které byly v rámci této práce implementovány:



Obrázek 6.5: Přeuspořádání paketů v burstu na základě klasifikace v HW.

1. Lineární průchod vstupním burstem a sestavení nového, kratšího burstu z paketů doposud neoznačených. Schéma této varianty je zobrazeno na obrázku 6.5. Vidíme, že první fází je detekce dosud neklasifikovaných paketů a jejich umístění do oddě-leného pole. V rámci jazyka C se jedná o pouhé přesunutí ukazatelů a nedochází tak ke kopírování dat. Takto oddělené pakety (na obrázku znázorněny jako N_i) jsou dále předány ke klasifikaci RTE ACL klasifikátoru, jehož výsledkem je pole indexů použitých pravidel. Z paketů, které již byly klasifikovány v síťové kartě (C_i), jsou extrahovány indexy pravidel a následně jsou obě skupiny indexů sloučeny do výsled-ného pole indexů (I_i). Další zpracování paketů pak probíhá stejným způsobem, jako v případě RTE ACL klasifikátoru (sekce 6.5).

2. Využití první položky ACL pravidla (viz sekce 4.4), která je zpracována přednostně. Do této položky je uložena informace o úspěšnosti HW klasifikace. Při detekci úspěš-nosti HW klasifikace pak RTE ACL klasifikátor skončí předčasně, jinak provede úplnou klasifikaci. Tento způsob klasifikace je zobrazen na obrázku 6.6. Vidíme, že ze vstupních paketů jsou nejprve extrahovány indexy pravidel získané klasifikací v síťové kartě. Příznak úspěšné klasifikace je pak připojen ke klasifikaci RTE ACL kla-sifikátorem. Tímto způsobem je získáno pole indexů pravidel (I_n) a další zpracování probíhá stejně jako u RTE ACL klasifikátoru (sekce 6.5).



Obrázek 6.6: Využití první jednobytové položky pro uložení příznaku klasifikace v HW.

6.7 Integrace

Hlavními implementačními výsledky této diplomové práce jsou dva optimalizované paketové klasifikátory. Jeden z nich je implementován čistě softwarově s využitím knihovny RTE ACL (sekce 6.5). Druhý z těchto klasifikátorů je pak navíc hardwarově akcelerován offloadem klasifikačních pravidel do síťové karty pomocí rozhraní RTE Flow (sekce 6.6). Kromě obou variant paketového klasifikátoru byly implementovány pomocné komponenty, které jsou navrženy obecně a mohou být použity pro implementaci dalších variant paketových klasifikátorů, případně hardwarovou akceleraci jiných komponent. Aby bylo možné v aplikaci DDoS Protector volit použitou implementaci klasifikátoru je využita abstraktní továrna (sekce 6.1).

V rámci integrace komponent implementovaných v rámci této práce bylo třeba upravit mitigační jádro tak, aby interně využívalo továrnu paketových klasifikátorů. Odkaz na takto vytvořený klasifikátor je dále předán oběma mitigačním oknům, kde jsou využity jednotlivými vlákny k samotné klasifikaci. Pro dealokaci paketového klasifikátoru, který již dále nebude potřeba, bylo přidáno volání destruktoru těsně po dealokaci obou mitigačních oken. Tímto způsobem je zaručeno, že klasifikátor při dealokaci není používán. Konkrétní varianta továrny paketových klasifikátorů je zvolena na základě nastavení v konfiguračním souboru, který je čten při spuštění aplikace DDoS Protector.

Kapitola 7

Dosažené výsledky

Funkcionalita komponent implementovaných v rámci této práce byla ověřena implementací jednotkových testů. Ty byly provedeny zvlášť pro jednotlivé komponenty uvedené v kapitole 6:

Testování *non_ov_ruleset* Komponenta realizující expanzi klasifikačních pravidel (sekce 6.2) byla testována nejpodrobněji. Pozitivní testy se zaměřují zejména na korektní sestavení pravidla, chybí-li nějaká ze sítí či rozsahů portů. Dále je ověřeno správné odstranění seznamů sítí a seznamů rozsahů portů, jehož výsledkem je rozšíření sady pravidel. Nejnáročnější jsou pak testy ověřující detekci průniků a překryvů mezi pravidly a následné vytvoření pseudopravidel.

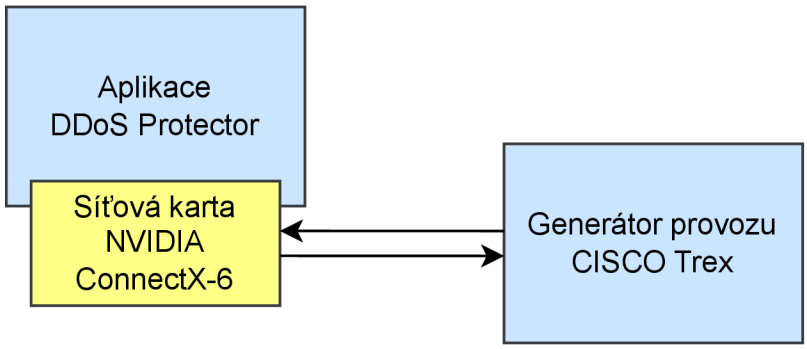
Kromě pozitivních testů byly vytvořeny i negativní testy. Ty se pak zaměřují na ověření chování komponent při nevalidních stavech, například při chybějící specifikaci cílové sítě nebo při chybně zadaných sítích (jedna síť IPv4, druhá IPv6).

Jednotkové testy této komponenty zahrnovaly celkem 16 testovacích případů.

Testování tabulky pseudopravidel Testy komponenty pro rychlé vyhledání původních pravidel z vygenerovaného pseudopravidla (sekce 6.3) se zaměřují zejména na ověření přidávání nových bloků indexů a jejich vyhledání. Klíčovou ověřovanou funkcionalitou je výpočet počtu bloků potřebných k uložení všech indexů pravidel. Sada jednotkových testů tabulky pseudopravidel zahrnuje 5 testovacích případů.

Testování RTE ACL klasifikátoru Ověření funkcionality samotného RTE ACL paketového klasifikátoru (sekce 6.5) spočívá zejména v korektním nastavení výstupních bitových masek jednotlivých pravidel, které značí indexy klasifikovaných paketů. Tyto testy byly implementovány jak pro klasifikaci paketů pouze jedním pravidlem, tak i pro klasifikaci více pravidly zároveň, což vynucuje vytvoření pseudopravidel a klasifikaci pomocí nich. Kromě úspěšné klasifikace byly otestovány i případy, kdy paket není klasifikován žádným pravidlem nebo vůbec žádné pravidlo instalováno nebylo. Celkově bylo pro komponentu RTE ACL paketového klasifikátoru implementováno 34 testovacích případů Data pro jednotkové testy byly vytvářeny ručně, a to záměrně s cílem ověřit konkrétní funkčnost.

Jednotkové testy aplikace DDoS Protector jsou navrženy tak, aby nebylo nutné používat síťovou kartu. Tento přístup však neumožňuje implementaci jednotkových testů pro žádnou z komponent využívající offloadu výpočtu na síťovou kartu. To se zejména týká komponenty *Flow Group Switcher* a RTE Flow klasifikátoru.



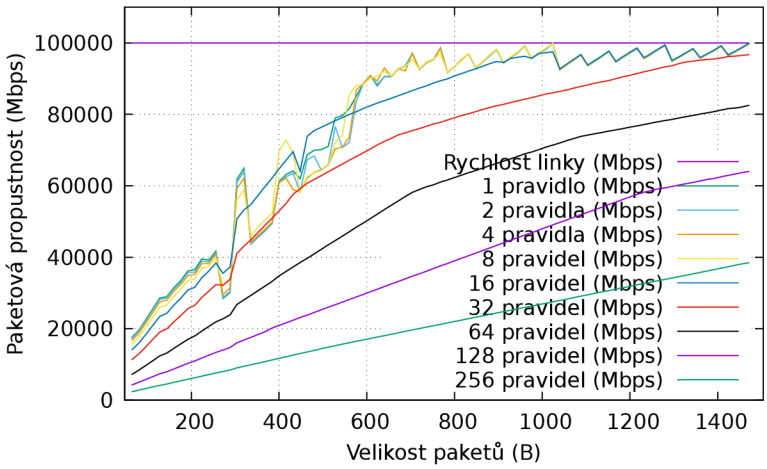
Obrázek 7.1: Testovací prostředí s generátorem provozu Cisco Trex

Kromě jednotkových testů byly ale implementované komponenty testovány integračními testy aplikace *DDoS Protector*. Tyto testy dopadly úspěšně jak u RTE ACL, tak i u RTE Flow klasifikátoru. Díky tomu je možné obě varianty použít v rámci aplikace DDoS Protector. Tyto integrační testy využívají mimo jiné i testovací prostředí zobrazené na obrázku 7.1. Vidíme, že v tomto prostředí je zapojen generátor provozu *Cisco Trex* [26]. Ten vytváří síťové toky o vysoké rychlosti (až 100 Gb/s) a provoz posílá do připojené síťové karty, která je ovládaná aplikací DDoS Protector. Aplikace pak zpracuje příchozí pakety a vrátí je zpět generátoru provozu, který může ověřit jejich korektní zpracování a také analyzovat propustnost aplikace DDoS Protector.

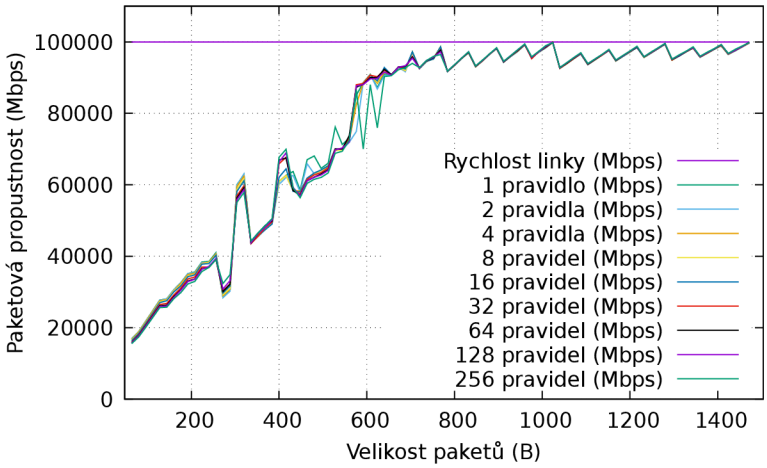
Analýza propustnosti zahrnovala srovnání optimalizovaných variant využívajících RTE ACL a RTE Flow s původním klasifikátorem. Toto srovnání probíhalo s různou velikostí paketů a různým počtem instalovaných pravidel. Jednotlivá klasifikační pravidla byla vytvořena tak, že IP adresy zdrojových i cílových sítí byly generovány náhodně, délka prefixu byla pak nastavena pevně na maximální délku. Porty transportní vrstvy zahrnovaly celý možný rozsah 0 až 65535. Tímto způsobem sice není při instalaci pravidel potřeba provádět předzpracování, to však na výkon klasifikace nemá vliv. Takto vytvořená sada 256 pravidel byla uložena do souboru a následně sdílena mezi jednotlivými běhy měření. Všechny varianty klasifikátoru tak byly testovány s pomocí stejné sady pravidel. Při konfiguraci nástroje *Trex* byla pak sada pravidel analyzována a nástroj byl konfigurován tak, aby alespoň část generovaných toků odpovídala instalovaným klasifikačním pravidlům. Aplikace DDoS Protector pak část generovaných paketů klasifikuje úspěšně a část neúspěšně.

Grafy uvedené na obrázku 7.2 zobrazují závislost propustnosti aplikace na velikosti paketů s využitím původního klasifikátoru (obrázek 7.2a) a klasifikátoru RTE ACL (obrázek 7.2b). Vidíme, že propustnost aplikace při malé velikosti paketů nedosahuje plné rychlosti linky, což je ale očekávané chování z důvodu vyšší režie. Naopak je vidět, že implementací RTE ACL klasifikátoru byla odstraněna závislost propustnosti aplikace na počtu instalovaných pravidel. Pro nejhorší případ 256 pravidel byla původní propustnost maximálně 40 Gb/s, zatímco při použití RTE ACL klasifikátoru bylo možné dosáhnout plné propustnosti 100 Gb/s pro pakety větší než 700 B.

Srovnání propustnosti mezi klasifikátorem využívajícím RTE ACL a dvěma implementovanými variantami RTE Flow klasifikátoru (sekce 6.6) je pak zobrazeno na obrázku 7.3. Vidíme, že všechny uvedené varianty dosahují téměř plného vytížení síťové linky už pro pakety větší než 700 B. Rozdíly mezi těmito variantami jsou ale velmi malé. U paketů, které jsou menší než 700 B je možné vidět, že offload do hardware přinesl zlepšení propustnosti. Naopak paketů větších než 700 B se ukazuje RTE ACL klasifikátor jako lepší. Důvod tohoto



(a) Propustnost aplikace bez využití upraveného klasifikátoru.

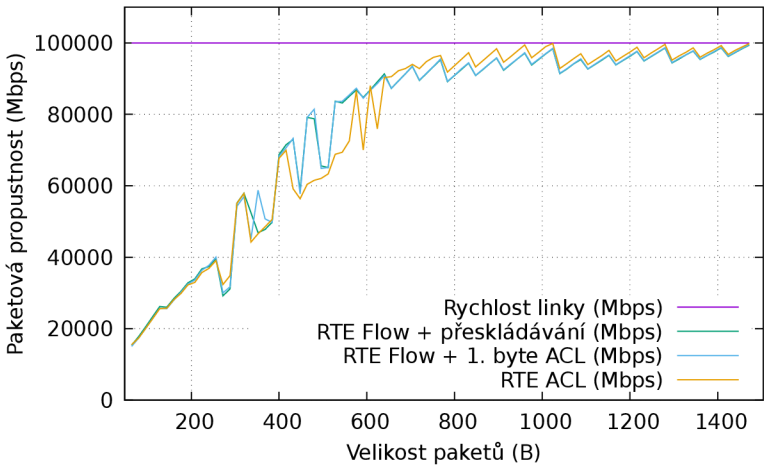


(b) Propustnost aplikace s klasifikátorem optimalizovaným pomocí RTE ACL.

Obrázek 7.2: Propustnost aplikace DDoS Protector s původním a RTE ACL klasifikátorem.

rozdílu není dostatečně prozkoumán, může se ale jednat o důsledek využití několika RTE Flow skupin, které zvyšují dobu zpracování jednotlivých paketů v hardware.

Rozdíl propustností aplikace mezi klasifikátory RTE ACL a RTE Flow je v grafu téměř zanedbatelný, proto bylo provedeno profilování nástrojem *perf* při plném vytížení aplikace testováním propustnosti. Doba strávená v této funkci je procentuálně vyjádřena tabulkou 7.1. Vidíme, že doba strávená v rámci paketového klasifikátoru klesla z původních 58 % na 22 % v případě použití RTE ACL klasifikátoru. Offloadování části klasifikačních pravidel do hardware pomocí RTE Flow přináší další, zhruba pětiprocentní, zlepšení výkonu, které se však již na propustnosti aplikace neprojevuje tak výrazně, protože mitigační komponenta již nezabírá tolik času, jako u neoptimalizované implementace. Rozdíl mezi dvěma implementovanými variantami RTE Flow klasifikátoru je téměř zcela zanedbatelný, protože varianta s přeskládáváním ukazatelů na pakety je jen o zhruba jedno procento lepší, než varianta s využitím prvního bytu RTE ACL klasifikačního pravidla.



Obrázek 7.3: Srovnání propustnosti aplikace s optimalizovanými klasifikátory při instalaci 256 pravidel.

Varianta	Relativní čas [%]
Původní	58.2
RTE ACL	22.2
RTE Flow, přeskládávání	16.8
RTE Flow, první byte ACL	18.1

Tabulka 7.1: Čas strávený klasifikační funkcí pro jednotlivé varianty klasifikátoru.

Z pohledu implementace a možností integrace do aplikace DDoS protector je pak srovnání dvou RTE Flow variant složitější. Varianta využívající přeskládávání byla snadno implementovatelná a je možné ji v budoucnu zobecnit zavedením delegace i na odlišné klasifikátory, než pouze RTE ACL. Toto zobecnění by naopak nemohlo být použito v případě druhé varianty využívající jednobytovou položku RTE ACL pravidla. Tuto variantu by ale bylo možné v budoucnu dále vylepšit například přesunem extrakce příznaku úspěšné klasifikace do komponenty *Packet Parser*, což by mohlo přinést další zlepšení výkonu.

Kapitola 8

Závěr

V rámci diplomové práce byla provedena analýza aplikace DDoS Protector, která je určena k potlačení útoků typu odepření služby. Na základě provedeného měření propustnosti a profilování pomocí existujících nástrojů byly identifikovány komponenty klíčové k optimalizaci a akceleraci této aplikace. Jako nejvýznamnější z pohledu stráveného času byl identifikován paketový klasifikátor, který vykazoval závislost propustnosti na počtu instalovaných pravidel.

V další fázi bylo navrženo několik možností optimalizace paketového klasifikátoru i dalších komponent pomocí prostředků dostupných ve frameworku DPDK. Jednou z navržených optimalizací byla implementace nového paketového klasifikátoru využívajícího knihovnu RTE ACL. Pro tuto komponentu bylo pak navrženo její rozšíření o možnost přenosu části klasifikačních pravidel do hardware síťové karty pomocí obecného rozhraní RTE Flow. Pro offload pravidel byly provedeny experimenty se síťovými kartami *Intel E810-CQDA2* a *NVIDIA ConnectX-6*. Karta od výrobce *Intel* však neměla dostatečnou podporu offloadu, a tak byly HW akcelerované komponenty vyvíjeny primárně pro síťovou kartu firmy *NVIDIA*.

Před samotnou implementací navržených komponent bylo nejprve nutné implementovat pomocné datové struktury a algoritmy, zejména mechanismus převodu mitigačních pravidel do korektního formátu a také způsob efektivní zpětné konverze do původní podoby. S využitím těchto komponent již bylo možné implementovat navržené klasifikační komponenty.

V poslední fázi bylo provedeno testování implementovaných součástí, a to jak z pohledu funkčnosti pomocí jednotkových a integračních testů, tak i z pohledu výkonnosti pomocí měření propustnosti a profilování aplikace v rámci testovacího prostředí aplikace DDoS Protector.

Z výsledků měření a profilování je pak zřejmé, že oproti původnímu klasifikátoru došlo ke značnému zlepšení výkonnosti, zejména v ohledu redukce závislosti propustnosti na počtu instalovaných pravidel. Například při instalaci 256 mitigačních pravidel se pro pakety o velikosti 700 B podařilo dosáhnout až pětinásobného zrychlení. Čas strávený v paketovém klasifikátoru byl pak redukován přibližně na jednu třetinu.

Jako budoucí rozšíření je možné provést optimalizaci algoritmů, které provádějí expanzi pseudopravidel. Tyto algoritmy nemají přímý vliv na výkon aplikace, protože se provádí pouze při změně instalované sady pravidel. Do budoucna je však vhodné brát v potaz i velmi rozsáhlé sady pravidel, jejichž expanze může trvat neúnosně dlouhou dobu, proto bude v rámci další práce nutné implementovat lepší variantu expanze sady pravidel.

Dalšími možným budoucím rozšířením může být úprava offloadované sady pravidel využívající spekulativního přístupu a RTE Flow akce *AGE*. Tato úprava by pak souvisela

i s detailnější analýzou skutečných možností jednotlivých síťových karet. Na základě této analýzy by mohly být navržené úpravy portovány i na síťové karty jiných výrobců.

Rozšíření také může zahrnovat úpravu testovacího prostředí, v rámci něhož je možné dále vylepšit jednotlivé testovací a profilovací případy, aby deterministicky měřily výkonnost aplikace, což by zahrnovalo i sestavení vlastních sad pravidel s různými parametry pro testování okrajových případů.

Literatura

[1] *DNS Amplification Attacks* [online]. Cybersecurity and Infrastructure security Agency, 2013-03-29 [cit. 2021-12-07]. Dostupné z: https://us-cert.cisa.gov/ncas/alerts/TA13-088A.

[2] *DPDK Programmer's Guide: 10. Mbuf Library* [online]. v21.11.0. [cit. 2022-05-03]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/mbuf_lib.html#mbuf-library.

[3] *DPDK Programmer's Guide: 12. Generic flow API (rte_flow)* [online]. v21.11.0. [cit. 2021-10-12]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/rte_flow.html.

[4] *DPDK Programmer's Guide: 27. Toeplitz Hash Library* [online]. v21.11.0. [cit. 2022-01-18]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/toeplitz_hash_lib.html.

[5] *DPDK Programmer's Guide: 43. Kernel NIC Interface* [online]. v21.11.0. [cit. 2021-11-29]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/kernel_nic_interface.html.

[6] *DPDK Programmer's Guide: 47. Packet Classification and Access Control* [online]. v21.11.0. [cit. 2021-10-19]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/packet_classif_access_ctrl.html.

[7] *DPDK Programmer's Guide: 7. Ring library* [online]. v21.11.0. [cit. 2021-11-29]. Dostupné z: http://doc.dpdk.org/guides/prog_guide/ring_lib.html.

[8] *DPDK Programmer's Guide: 9. Mempool Library* [online]. v21.11.0. [cit. 2021-09-27]. Dostupné z: https://doc.dpdk.org/guides/prog_guide/mempool_lib.html.

[9] *DPDK Testpmd Application User Guide* [online]. v21.11.0. [cit. 2021-05-03]. Dostupné z: https://doc.dpdk.org/guides/testpmd_app_ug/index.html.

[10] *DPDK Testpmd Application User Guide: 4.13. Flow rules management* [online]. v21.11.0. [cit. 2021-05-03]. Dostupné z: https://doc.dpdk.org/guides/testpmd_app_ug/testpmd_funcs.html#flow-rules-management.

[11] *Mēris botnet, climbing to the record* [online]. [cit. 2021-12-06]. Dostupné z: https://blog.qrator.net/en/meris-botnet-climbing-to-the-record_142/.

[12] *NAPI* [online]. The Linux Foundation [cit. 2021-09-27]. Dostupné z: https://wiki.linuxfoundation.org/networking/napi.

[13] *UDP-Based Amplification Attacks* [online]. Cybersecurity and Infrastructure security Agency, 2014-01-17 [cit. 2021-12-07]. Dostupné z: https://us-cert.cisa.gov/ncas/alerts/TA14-017A.

[14] *Universal TUN/TAP device driver* [online]. The kernel development community [cit. 2021-11-29]. Dostupné z: https://www.kernel.org/doc/html/latest/networking/tuntap.html.

[15] Benvenuti, C. *Understanding Linux network internals*. Sebastopol, CA, USA: O´Reilly, 2006. ISBN 0-596-00255-6.

[16] Burakov, A. Memory in Data Plane Development Kit Part 1: General Concepts. [online]. Intel Corporation. 2019, [cit. 2021-09-27]. Dostupné z: https://software.intel.com/content/www/us/en/develop/articles/memory-in-dpdk-part-1-general-concepts.html.

[17] Corbet, J. *Linux device drivers*. 3rd ed. Sebastopol, CA, USA: O´Reilly, 2005. ISBN 0-596-00590-3.

[18] Douligeris, C. a Mitrokotsa, A. DDoS attacks and defense mechanisms: a classification. In: *Proceedings of the 3rd IEEE International Symposium on Signal Processing and Information Technology (IEEE Cat. No.03EX795)*. 2003, s. 190–193. DOI: 10.1109/ISSPIT.2003.1341092.

[19] Goldschmidt, P. a Kučera, J. Defense Against SYN Flood DoS Attacks Using Network-based Mitigation Techniques. In: *2021 IFIP/IEEE International Symposium on Integrated Network Management (IM)*. 2021, s. 772–777.

[20] Gorman, M. *Understanding the Linux Virtual Memory Manager*. Upper Saddle River, NJ, USA: Pearson Education, Inc., 2004. ISBN 0-13-145348-3.

[21] Gregg, B. Flame Graphs. [online]. 2020, [cit. 2022-05-03]. Dostupné z: https://www.brendangregg.com/flamegraphs.html.

[22] Gupta, B. a Badve, O. Taxonomy of DoS and DDoS attacks and desirable defense mechanism in a Cloud computing environment. In: *Neural Computing and Applications*. 2017, sv. 28, s. 3655–3682. DOI: 10.1007/s00521-016-2317-5.

[23] Gutnikov, A., Kupreev, O. a Shmelev, Y. *DDoS attacks in Q3 2021* [online]. [cit. 2021-12-07]. Dostupné z: https://securelist.com/ddos-attacks-in-q3-2021/104796/.

[24] Puš, V. *Packet Classification Algorithms*. 2012. Disertační práce. Vysoké učení technické v Brně. Fakulta informačních technologií.

[25] Rojas Cessa, R. *Interconnections for Computer Communications and Packet Networks*. Boca Raton: CRC Press, 2017. ISBN 1482226960.

[26] Tran, D. *Prostředí pro testování zařízení umožňujících ochranu před DoS útoky*. 2020. Diplomová práce. Vysoké učení technické v Brně. Fakulta informačních technologií.

[27] Viktorin, J. *Akcelerační model DCPro DPDK*. Listopad 2020. Interní dokument [Online] [cit. 2022-01-24]. Dostupné z: https://redmine.liberouter.org/issues/3100.

[28] Zargar, S. T., Joshi, J. a Tipper, D. A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks. *IEEE Communications Surveys Tutorials*. 2013, sv. 15, č. 4, s. 2046–2069. DOI: 10.1109/SURV.2013.031413.00127.

[29] Zhu, H. *Data Plane Development Kit (DPDK): A Software Optimization Guide to the User Space-Based Network Applications*. Milton Park, UK: Taylor & Francis Group, 2021. ISBN 0367520176.

Příloha A

Experimenty s RTE Flow pomocí nástroje *testpmd*

A.1 Síťová karta Intel E810-CQDA2

```
testpmd> flow create 0 ingress pattern end actions drop / end
ice_flow_create(): Failed to create flow
port_flow_complain(): Caught PMD error type 2 (flow rule (handle)):
No memory for PMD internal items: Invalid argument

testpmd> flow create 0 ingress pattern eth / ipv4 / end actions drop / end
ice_flow_create(): Failed to create flow
port_flow_complain(): Caught PMD error type 10 (item specification):
cause: 0x7ffcb47a0218, Invalid input set: Invalid argument

testpmd> flow create 0 \
    ingress pattern eth \
        src spec 00:00:00:00:00:00 src mask 00:00:00:00:00:00 \
        dst spec 00:00:00:00:00:00 dst mask 00:00:00:00:00:00 / \
    ipv4 \
        src spec 0.0.0.0 src mask 0.0.0.0 \
        dst spec 0.0.0.0 dst mask 0.0.0.0 / \
    end actions drop / end
ice_flow_create(): Failed to create flow
port_flow_complain(): Caught PMD error type 10 (item specification):
cause: 0x7ffcb47a0218, Invalid input set: Invalid argument
```

Výpis A.1: Ovladač neumožňuje vytvářet výchozí pravidla klasifikující všechny pakety

```
testpmd> flow create 0 group 0 priority 1 ingress \
    pattern eth / ipv4 src spec 10.0.0.0 src mask 255.255.0.0 / end \
    actions drop / end
ice_flow_create(): Failed to create flow
port_flow_complain(): Caught PMD error type 4 (priority field):
cause: 0x7ffcb47a0198, Not support priority.: Invalid argument
```

Výpis A.2: Chyba při zadání prioritní úrovně jiné, než 0.

```
testpmd> flow create 0 group 1 ingress \
    pattern eth / ipv4 src spec 10.0.0.0 src mask 255.255.0.0 / end \
    actions drop / end
ice_flow_create(): Failed to create flow
port_flow_complain(): Caught PMD error type 3 (group field):
cause: 0x7ffcb47a0198, Not support group.: Invalid argument
```

Výpis A.3: Chyba při zadání skupiny jiné, než 0.

A.2 Síťová karta NVIDIA ConnectX-6 Dx 100GbE

```
testpmd> flow create 0 ingress \
    pattern eth / ipv4 / tcp src spec 80 src last 89 / end \
    actions drop / end
port_flow_complain(): Caught PMD error type 13 (specific pattern item):
cause: 0x7ffe69044428, range is not valid: Invalid argument
```

Výpis A.4: Chyba při vytvoření pravidla definujícího rozsah TCP portů.

```
testpmd> flow create 0 priority 4 \
    ingress pattern eth / ipv4 / tcp / end \
    actions drop / end
port_flow_complain(): Caught PMD error type 4 (priority field):
priority out of range: Operation not supported
```

Výpis A.5: Chyba při specifikaci priority vyšší, než 3.

```
testpmd> flow shared_action 0 create ingress action jump group 1 / end
Shared action #0 destroyed
port_flow_complain(): Caught PMD error type 16 (specific action):
action type not supported: Operation not supported
```

Výpis A.6: Chyba pri specifikaci *indirect* akce typu *jump*. Pozn.: v testované verzi DPDK je *indirect* akce nazvaná jako *shared*.

```
# Vytvoření původního pravidla.
testpmd> flow create 0 priority 3 ingress \
    pattern end \
    actions jump group 1 / end
Flow rule #0 created
# Vytvoření pravidla s~vyšší prioritou a jinou cílovou skupinou skoku.
testpmd> flow create 0 priority 2 ingress \
    pattern end \
    actions jump group 2 / end
Flow rule #1 created
# Odstranění původního pravidla
testpmd> flow destroy 0 rule 0
Flow rule #0 destroyed
# Kopie pravidla #1, avšak s~nižší prioritou.
testpmd> flow create 0 priority 3 ingress \
```

```
    pattern end \
    actions jump group 2 / end
Flow rule #2 created
# Odstranění pravidla s~vyšší prioritou.
testpmd> flow destroy 0 rule 1
Flow rule #1 destroyed
```

Výpis A.7: Atomické nahrazení RTE Flow pravidla pomocí více priorit