University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Denis Salopek

HYBRID HARDWARE/SOFTWARE DATAPATH FOR NEAR REAL-TIME RECONFIGURABLE HIGH-SPEED PACKET FILTERING

DOCTORAL THESIS

Zagreb, 2022

University of Zagreb

FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

Denis Salopek

HYBRID HARDWARE/SOFTWARE DATAPATH FOR NEAR REAL-TIME RECONFIGURABLE HIGH-SPEED PACKET FILTERING

DOCTORAL THESIS

Supervisor:
Associate Professor Miljenko Mikuc, PhD

Zagreb, 2022

Sveučilište u Zagrebu

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Denis Salopek

HIBRIDNA SKLOPOVSKO/PROGRAMSKA PODATKOVNA STAZA ZA BRZO FILTRIRANJE PAKETA REKONFIGURABILNA U PRIBLIŽNO STVARNOM VREMENU

DOKTORSKI RAD

Mentor:
izv. prof. dr. sc. Miljenko Mikuc

Zagreb, 2022.

Doktorski rad izrađen je na Sveučilištu u Zagrebu, Fakultetu elektrotehnike i računarstva, na Zavodu za telekomunikacije.

Mentor:
izv. prof. dr. sc. Miljenko Mikuc

Doktorski rad ima: 102 stranice

Doktorski rad br.: ___________

Mentor's Curriculum Vitae

Miljenko Mikuc is an Associate Professor at the University of Zagreb, Faculty of Electrical Engineering and Computing, Department of Telecommunications. He graduated in 1987 and received his PhD in the field of technical sciences, electrical engineering in 1997 from the University of Zagreb, Faculty of Electrical Engineering and Computing (FER).

He has participated on seven scientific projects of the Ministry of Science, Education and Sports of the Republic of Croatia and on the international project "Verification and validation methods for formal descriptions (COST 247)". He participated and led projects with "The Boeing Company — IDS, LabNet Analysis, Modeling Simulation and Experimentation", "International Computer Science Institute / University of California, Berkeley", "The FreeBSD Foundation", multi-year research project in the field of information and communication technology with Ericsson Nikola Tesla d.d., "Customized IMUNES for Ericsson (E-IMUNES)" and IRI-projects "New Generation Lawful Interception System — NG LI" and "Threat monitoring platform for heterogeneous network environments — SOC4.0" where FER is a partner of the company SedamIT d.o.o.

At the University of Zagreb, Faculty of Electrical Engineering and Computing, he is responsible for the following undergraduate and graduate courses: "Digital Logic", "Network Programming", "Internet Security", "Computer Security" and "Network and Services Management." At the postgraduate doctoral study program, he is responsible for "Formalisms in Telecommunications" and "Communication Protocols — Selected Topics". As a mentor, he successfully led more than 200 undergraduate and graduate students, while at the postgraduate study he was a mentor for 15 master theses and 3 doctoral theses.

He has published over 40 scientific and professional papers in journals and conference proceedings in the field of communication networks, protocols, virtualization, formal methods and security. He is a member of a professional association of IEEE. He participates in the work of the Technical Program Committee of the International Scientific Conference of SoftCOM and as a reviewer at a number of international conferences.

Životopis mentora

Miljenko Mikuc je izvanredni profesor na Zavodu za telekomunikacije Fakulteta elektrotehnike i računarstva Sveučilišta u Zagrebu. Diplomirao je 1987. godine a doktorsku disertaciju iz područja tehničkih znanosti, polja elektrotehnika, "Postupci provjere ispravnosti specifikacije telekomunikacijskih procesa" obranio je 1997. godine.

Sudjelovao je kao istraživač na sedam znanstvenih projekata Ministarstva znanosti, obrazovanja i sporta Republike Hrvatske te na međunarodnom projektu "Verification and validation methods for formal descriptions (COST 247)". Bio je voditelj dva projekta Primjene informacijske tehnologije pod pokroviteljstvom Ministarstva znanosti i tehnologije Republike Hrvatske. Bio je voditelj projekata suradnje s "The Boeing Company — IDS, LabNet Analysis, Modeling Simulation and Experimentation", "International Computer Science Institute", "The FreeBSD Foundation" iz SAD-a te višegodišnjeg istraživačkog projekta u sklopu suradnje na području informacijskih i komunikacijskih tehnologija s kompanijom Ericsson Nikola Tesla d.d., "Prilagođen IMUNES za Ericsson (E-IMUNES)", IRI-projekta "Nova generacija rješenja za zakonsko presretanje podataka — NG LI". Voditelj je IRI-projekta "Platforma za nadzor ugroza u heterogenim mrežnim okruženjima — SOC4.0." na kojem je FER partner tvrtke prijavitelja SedamIT d.o.o.

Sudjeluje u nastavi na preddiplomskom i diplomskom studiju kao nositelj ili su-nositelj predmeta "Digitalna logika", "Mrežno programiranje", "Sigurnost u Internetu", "Sigurnost računalnih sustava" i "Upravljanje mrežom i uslugama". Na poslijediplomskom studiju sunositelj je predmeta "Formalizmi u telekomunikacijama" i "Odabrana poglavlja komunikacijskih protokola". Pod njegovim mentorstvom uspješno je završilo studij više od 200 studenata na preddiplomskom i diplomskom studiju. Na poslijediplomskom studiju bio je mentor pri izradi 15 magistarskih radova te 3 doktorska rada.

Objavio je preko 40 znanstvenih i stručnih radova u časopisima i zbornicima konferencija u području komunikacijskih mreža, protokola, virtualizacije, formalnih metoda i sigurnosti. Član je stručne udruge IEEE. Sudjeluje u radu tehničkog programskog odbora međunarodne znanstvene konferencije SoftCOM, te kao recenzent na većem broju međunarodnih konferencija.

Acknowledgments

Prof. Mikuc, a great person and advisor, provided an incredible amount of patience and assistance. He has been the driving force behind my efforts, and I owe him a debt of gratitude because without him, I would have lost this battle several times over the past few years.

Thank you, Marko Zec, for completing your own dissertation and taking the time to help me lay the groundwork for my work and inspire me to finish it now or never. Thank you, Valter Vasić for getting me into this mess all those years ago, and for your unwavering (technical and psychological) support and words of encouragement.

Thank you to all my friends, co-workers, and acquaintances who helped me in any way.

And of course, I want to express my great gratitude to my Marina: for all her patience, support, and love that kept me sane.

This quest is over, what's next?

Abstract

The increasing number of volumetric Distributed Denial-of-Service (DDoS) attacks, as well as their intensity and scale, have led many security experts to research and work on solutions to protect against these types of attacks. Although solutions to combat such attacks already exist, they are typically based on expensive and inflexible network equipment or on the (half-true) assumption that software filters running on commodity hardware are incapable of handling high-speed traffic and delivering sufficient throughput. The idea of combining the best of both worlds (hardware speed and software versatility) is found in a number of solutions, but cannot prevail against massive DDoS attacks with millions of attackers, as such solutions often rely on rulesets with a large number of IP prefixes used with a rule-by-rule packet filtering paradigm.

This thesis presents and evaluates a hybrid hardware / software packet filter prototype as a method for mitigating volumetric DDoS attacks using a NetFPGA SUME prototyping board and a high-performance, high-speed, reduced feature-set software packet filter. It demonstrates a novel approach to offload the filtering rules (or parts of them) to the hardware by taking advantage of a modern Longest Prefix Matching (LPM) algorithm to utilize allowlists and blocklists for protection against millions of IP prefixes. The results of this work show that this type of filtering can be performed in high-speed network environments using a single CPU core. The system architecture is designed to allow scaling to much higher throughput.

The results of this thesis show improvements over software-only filtering of up to nearly 30%, depending on the combination of rulesets used, the offloading methods, and the type of traffic filtered. The components of the hybrid filter can be implemented on commodity hardware and provide an alternative to expensive or less effective filters. Developing a system that combines fast DDoS detection (with low response times) and this type of filtering could provide high-speed protection against volumetric DDoS attacks. Internet Service Providers (ISPs) and datacenters could take advantage of such filtering methods without being harmed by DDoS attacks or having to compromise the privacy of their data by outsourcing filtering to third parties. Due to the low cost of the commodity, off-the-shelf hardware that these filters use, they can also be deployed by small or medium-sized businesses.

Keywords: hybrid filters, DDoS mitigation, FPGA, hardware / software packet processors, filter performance

HIBRIDNA SKLOPOVSKO/PROGRAMSKA PODATKOVNA STAZA ZA BRZO FILTRIRANJE PAKETA REKONFIGURABILNA U PRIBLIŽNO STVARNOM VREMENU — prošireni sažetak

Porast broja volumetrijskih distribuiranih napada uskraćivanjem resursa (DDoS napada), kao i njihovog intenziteta i razmjera, već dugo potiču stručnjake iz područja sigurnosti na istraživanje i rad na rješenjima za zaštitu od njih. Iako postoji niz rješenja za borbu protiv takvih napada, oni su obično temeljeni na skupoj i nefleksibilnoj sklopovskoj mrežnoj opremi ili pretpostavci da programski filtri koji rade na računalima opće namjene nisu sposobni obraditi promet pri velikim brzinama i tako osigurati dovoljno veliku propusnost. Ideja o kombiniranju najboljeg iz oba svijeta (sklopovska brzina i svestranost programskog rješenja pristupa) također se nalazi u brojnim rješenjima, ali u obliku u kojem se ne mogu boriti protiv masivnih DDoS napada s milijunima napadača jer se često oslanjaju na skupove pravila s velikim brojem IP prefiksa i paradigmom filtriranja paketa pravilo po pravilo.

U ovoj disertaciji predstavlja se i ispituje implementacija prototipa hibridnog sklopovsko / programskog filtra paketa za ublažavanje volumetrijskih DDoS napada pomoću razvojne pločice za prototipiziranje mrežnih funkcija NetFPGA SUME i jednostavnog programskog filtra. Predlaže se novi pristup za rasterećivanje pravila (ili dijelova pravila) za filtriranje na sklopovlju koristeći moderni LPM algoritam i popise dopuštenih i nedopuštenih IP prefiksa za zaštitu i od više milijuna napadača. Disertacija prikazuje rezultate filtriranja izmjerene u 10G mreži. Mjerenja su izvedena na jednoj jezgri procesora, ali s mogućnošću skaliranja na više jezgri za mnogo veću propusnost filtra.

Hibridni podatkovni put predstavljen u ovoj disertaciji prikazuje poboljšanja u odnosu na podatkovni put bez sklopovskog rasterećivanja. Ovakav sustav može filtrirati pakete s visokom propusnošću, osobito kada se koristi LPM za spremanje i pretraživanje IP adresa. Rezultati prikazuju poboljšanja u rasponu do oko 30%, ovisno o kombinaciji skupova pravila, metoda rasterećivanja i vrsti prometa koji dolazi do filtra. Komponente hibridnog filtra mogu se implementirati na računalima opće namjene i pružaju alternativu skupim sklopovskim ili manje učinkovitim programskim filtrima. Pružatelji pristupa Internetu (ISP), mala i srednja poduzeća te podatkovni centri mogli bi iskoristiti navedene metode filtriranja bez rizika štete od DDoS napada ili ugrožavanja privatnosti svojih podataka prenošenjem odgovornosti filtriranja na treće strane.

Poglavlje 1 — Uvod i motivacija

Prvo poglavlje uvodi u problematiku istraživanja, te opisuje motivaciju i ciljeve istraživanja. Obrada paketa u mrežama s visokom propusnošću uglavnom je posao specijaliziranih mrežnih uređaja temeljenih na sklopovlju koje bez poteškoća, ali uz svoje nedostatke, mogu dovoljno brzo klasificirati i filtrirati pakete. Da bi se taj posao na mreži brzine 100 Gbit/s mogao obavljati programski, potrebno je koristiti filtriranje koje podržava propusnost od više od 148 milijuna paketa u sekundi (engl. millions of packets per second — Mpps). Na računalu opće namjene s primjerice brzinom takta procesora od 4 Ghz, to bi značilo da svaki paket treba obraditi u manje od 27 ciklusa takta.

Ovisno o tipu uređaja, klasificiranje paketa se obavlja po nekom kriteriju, npr. pretraga po odredišnoj MAC adresi ili VLAN oznaci okvira kod preklopnika, ili po izvorišnoj / odredišnoj IP adresi paketa kod usmjeritelja i vatrozida. Obavljanje takvih provjera vrlo je teško izvesti u manje od potrebnih 27 ciklusa takta, budući da se u to vrijeme mora moći obaviti više operacija ili dohvaćanja iz memorije. Navedeni primjer s 27 ciklusa odnosi se na promet s najmanjim veličinama okvira, kad je uređaju "najteže" raditi budući da tad dolaze s najkraćim vremenom međudolazaka, ali sva oprema u mrežnoj infrastrukturi mora moći uvijek obraditi sve veličine paketa pri svim brzinama. U suprotnom, ako dođe do kvara na jednom uređaju, tada raspoloživost cijele infrastrukture više nije zajamčena.

Tu činjenicu iskorištavaju maliciozni korisnici na Internetu te napadima uskraćivanja usluge (engl. Denial of Service — DoS) i raspodijeljenim napadima uskraćivanja usluge (engl. Distributed DoS — DDoS) pokušavaju onesposobiti korisnicima pristup nekim uslugama. DDoS napadi sve su češći, a nastaju tako da zaražena računala pod nadzorom napadača ("bot" računala) šalju promet na žrtvu i tako zauzimaju resurse regularnim korisnicima, a budući da takvih računala može biti i više milijuna, od takvog napada se vrlo teško obraniti. Ovo istraživanje se bavi samo IPv4 prometom jer IPv6 promet još nije toliko zastupljen u Internetu, niti se u bliskoj budućnosti očekuje da će IPv4 potpuno nestati, pa je tako i DDoS napada s IPv4 adresama puno više.

Na kraju poglavlja prikazana je struktura rada i opis pojedinih poglavlja.

Poglavlje 2 — Zaštita od DDoS napada

Drugo poglavlje opisuje trenutno stanje zaštite od DDoS napada, a koriste se tri tipa pristupa zaštiti: delegiranje trećim stranama, zaštita uređajima u infrastrukturi čuvane mreže (on-site), te kombinacijom ta dva pristupa. Delegiranje trećim stranama radi se tako da se sav promet preusmjerava servisima za zaštitu od DDoS napada koji zatim po potrebi "čiste" od opasnog i sumnjivog takvog prometa te prosljeđuju mreži kojoj je promet i namijenjen. Ovakvo preusmjeravanje prometa donosi potencijalne probleme ako je promet osjetljiv na čak i mala kašnjenja ili sadrži osjetljive i privatne informacije za koje nije prikladno da im mogu pristupiti treće strane (kao npr. u financijskom sektoru).

Zaštita na licu mjesta (engl. *on-site*) radi se uređajima koji mogu filtrirati promet pomoću specijaliziranog sklopovlja (hardware), programski (software) ili hibridno (kombinacija sklopovlja i programskog načina). Sklopovsko filtriranje obavlja se uređajima specifično izrađenima za takvu zaštitu, s visokom propusnošću, ali i visokim cijenama godišnjih licenci za korištenje pripadajuće programske podrške. Osim cijene, negativne strane ovakvih uređaja uključuju nedostatak fleksibilnosti, kao i kompleksnost pri modificiranju ili ažuriranju, zbog čega već nakon nekoliko godina prestaju podržavati zahtjeve trenutnih brzina mreže pa ih je potrebno mijenjati novijim i skupljim modelima. Korištenje TCAM tehnologije u ovakvim uređajima dodatno dovodi do toga da su oni veliki potrošači električne energije što je još jedna njihova negativna strana. Ostale tehnologije koje se koriste pri ovom filtriranju su ASIC i FPGA, pri čemu ASIC ima slične nedostatke kao i TCAM, a FPGA se odvaja od njih po tome što ima mogućnost reprogramiranja.

Posljednjih godina se pojavljuju programski okviri (engl. *framework*) za brzu obradu paketa na računalima opće namjena koji uz dovoljno napredno sklopovlje u njima mogu postizati rezultate procesiranja paketa slične sklopovskim filtrima. To su Netmap, DPDK i XDP/eBPF, svaki sa svojim prednostima i nedostatcima, od kojih je svima zajednička fleksibilnost i kontrola nad filtrima stvorenima pomoću njih, budući da su jednostavniji i programabilni, za razliku od većine sklopovskih sustava.

Hibridna zaštita spaja sklopovsku s programskom zaštitom i najčešće koristi neku vrstu sklopovlja da bi djelomično (ili potpuno) preuzeo filtriranje na sebe i tako "olakšao" posao filtriranja programskoj podršci za koju se očekuje da je slabijih performasi.

Zaštiti od DDoS napada pristupa se na način sličan standardnom vatrozidu — stvaraju se liste pravila s različitim poljima koja se provjeravaju (kao npr. izvorišna ili odredišna IP adresa, protokol ili vrata transportnog sloja) i svaki paket prolazi kroz te liste te se zaglavlje svakog paketa uspoređuje sa zadanim poljima. U nekim slučajevima ovakve liste pravila pokušavaju se reducirati tako da budu minimizirane (tj. jednostavnije), ili se pomoću alata za klasifikaciju paketa (engl. *PCE — packet classification engine*) pokušava doći do metode kojom se mogu u što manje koraka dohvatiti iz memorije. Takvi alati i dalje za pretpostavku imaju da je zaštita od DDoS napada moguća samo uz pomoć velikog broja odvojenih pravila, a za velike volumetrijske DDoS napade potrebni su deseci tisuća ili čak milijuni takvih zapisa.

Osim navedenih "aktivnih" obrana od DDoS napada, postoji i *blackhole* usmjeravanje, gdje se žrtvina odredišna IP adresa može prijaviti mrežnom pružitelju usluga te on sav promet koji ide na tu IP adresu preusmjerava u "crnu rupu", tj. odbacuje. Time se štiti ostatak mreže jer se odbacivanjem velike količine štetnog prometa štedi na propusnosti, ali i efektivno ispunjava cilj samih napadača jer je žrtva od tog trenutka nedostupna ostatku korisnika.

U ovom poglavlju još se opisuje podloga za ovaj rad iz prethodnog istraživanja — način na koji se poboljšava postojeće filtriranje u službi obrane od volumetrijskih DDoS napada zamjenom velikih popisa pravila s manjim popisom pravila, ali korištenjem dodatnih tablica u kojima se spremaju IP adrese ili podmreže (npr. propusne ili blokirajuće), a koje se iz takvih tablica puno brže dohvaćaju najduljim prefiksnim podudaranjem (engl. LPM — *longest prefix matching*). Predlaže se i hibridno filtriranje uz pomoć sklopovlja i programske podrške koncipirano na ovakvom LPM pretraživanju, pri čemu se odmiče od paradigme da je za obranu od DDoS napada potrebno održavati monolitne liste kompleksnih pravila.

Poglavlje 3 — Model hibridnog sustava

U trećem poglavlju opisan je model podatkovnog puta opisanog hibridnog filtra i objašnjeni su razlozi za odabirom pojedinih komponenata: NetFPGA SUME za sklopovski dio i programski filtar razvijan u prethodnim istraživanjima i projektima kao programski dio. Opisane su prepreke pri implementaciji ovako opisanog modela te je prikazan i novi model koji zaobilazi nedostatke prvotnoga. Opisano je na koji se način pojedine komponente modela izmjenjuju tako da je moguće zaobići opisane prepreke. Glavni razlog za promjenom modela je nemogućnost odabranog sklopovlja da na ovakav način radi kao mrežno sučelje pri dovoljno velikim brzinama i pokušaj autora da implementira verziju koja će biti dovoljno funkcionalna, gdje se na kraju pokazalo da na postojećem sklopovlju to nije moguće bez značajnijih i kompleksnih promjena.

Paketi koji se trebaju filtrirati prolaze prvo kroz sklopovlje koje ih parsira, filtrira i stvara meta-podatke ako ih je potrebno proslijediti programskom filtru. Programski filtar dobiva iste pakete nadograđene s meta-podacima, i ovisno o tome kako je programiran, parsira paket i meta-podatke, te izvršava potrebnu akciju.

Nadalje, opisana su pravila koja se koriste u korištenom filtru i kategorizirana su na način koji omogućava pregled kako se filtriranje (ili dio filtriranja) za pojedinu kategoriju može odraditi na sklopovlju ili u programskom filtru. Pravila su izgrađena od jedne akcije i jednog ili više uzorka (odvojivi osnovni dijelovi). Akcija može biti terminirajuća (ako se nakon pravila prekida daljnji pregled paketa: npr. ACCEPT ili DENY) ili neterminirajuća (ako se nastavlja provjera pravila). Osim toga, akcija može biti brojuća (ako treba javiti programskom dijelu da je pravilo pogođeno) ili nebrojuća (ako za pravilo nije potrebno povećavati brojač). Uzorci su podijeljeni u tri vrste: oni koji mogu biti potpuno ili djelomično odrađeni u sklopovlju, te oni koji ne mogu biti odrađeni u sklopovlju.

Zbog ovih podjela, jedno pravilo može biti podijeljeno po tome kako ga je moguće odraditi u sklopovlju: potpuno, djelomično ili nemoguće, s time da je potpuno rasterećivanje moguće na pet načina (od čega su dva načina jednaka sa stajališta sklopovlja), a djelomično na tri načina. Tako se dolazi do ukupno osam kategorija pravila s ukupno pet različitih tipova informacija koje je potrebno komunicirati između sklopovlja i programskog dijela: za uzorke koji se djelomično odrađuju u sklopovlju, za uzorke koji se u potpunosti odrađuju u sklopovlju, za sva pravila koja su brojeća, za pravila koja su terminirajuća te dodatna informacija o tome je li bilo koje pravilo u sklopovlju pogođeno ili nije.

Poglavlje 4 — Implementacija filtra paketa

U četvrtom poglavlju prikazan je primjer ovakvog filtra koji bi dobro iskoristio LPM pretragu IP adresa i podmreža protiv DDoS napada čak i s više milijuna različitih napadača, korištenjem više različitih tablica (popisa IP adresa i podmreža koje su sigurne za automatsko propuštanje, ili maliciozne za automatsko blokiranje). Rad filtra je prikazan u više različitih verzija, ovisno o statusu sigurnosti mreže. Kad mreža nije u pod DDoS napadom, filtar propušta pakete s adresama iz tablice administratorskih IP adresa, blokira ranije prikupljene zlonamjerne adrese, te prati neke od predefiniranih sumnjivih adresa na samom početku (npr. iz područja sumnjivih odnosa). Nakon toga propušta ranije prikupljene sigurne adrese i potencijalno adrese s nešto manjom garancijom sigurnosti (npr. iz zemalja u kojoj se nalazi mreža) i na kraju propušta i prati količinu prometa svih ostalih. Ako se uz filtar koristi i dodatni alat za prepoznavanje DDoS napada s mogućnošću izoliranja napadačkih IP adresa, njemu bi se mogli prosljeđivati uzorci potrebnog prometa da on donese odluku o koracima koji slijede ako se dogodi DDoS napad. Budući da je za prikupljanje malicioznih adresa sa nekom dozom sigurnosti potrebno određeno vrijeme, u tom slučaju bi filtar mogao prijeći u način rada gdje filtrira i dalje sve kao i prije, ali sav nekategorizirani promet blokira i tako osigurava očuvanje propusnosti unutar štićene mreže. Moguće je i prijeći u restriktivniji način rada gdje se propušta isključivo promet s visokom garancijom sigurnosti (s adresama prikupljenima ranije). Kad sustav detekcije odredi maliciozne adrese, prosljeđuje ih filtru koji ih dodaje u tablicu za blokiranje te se njegov rad može vratiti u normalno stanje (prije otkrivanja DDoS napada), s pretpostavkom da je većina malicioznog prometa blokirana na početku filtriranja.

Opisane su sklopovska i programska implementacija hibridnog sustava i model distributora posla koji određuje kako filtriranje razdijeliti na sklopovski i programski dio, te "dogovoriti" komunikaciju između ta dva dijela. Razlog odabira FPGA za tehnologiju sklopovskog dijela je njena fleksibilnost zbog mogućnosti višestrukog rekonfiguriranja ali i iskorištavanje paralelizma koji FPGA donosi. NetFPGA SUME je razvojna pločica za prototipiziranje mrežnih funkcija za brze mreže koja se od 2015. godine koristi za velik broj istraživanja u raznim projektima te pruža mogućnosti za prototipiziranje ovakvog filtra pri 10G brzinama.

U sklopovskoj implementaciji prikazan je podatkovni put prototipa hibridnog filtra u NetFPGA SUME razvojnoj pločici, koristeći protokol AXI4-Stream za komunikaciju između pojedinih modula u "cjevovodu" sustava. "Cjevovod" je izgrađen od serijski ili paralelno spojenih modula i sastoji se od dva dijela: jednim dijelom putuju paketi koji dolaze s dolaznog mrežnog sučelja, tj. oni koji se provjeravaju (filtriraju) pa proslijeđuju na izlazno sučelje ako je potrebno, te tzv. kontrolni paketi kojima se regulira interna logika unutar sklopovlja (npr. postavljanje vrijednosti memorije ili uključivanje i isključivanje pojedinih parsera). Iz "pravih" paketa se izdvajaju svi potrebni podatci koji su potrebni za filtriranje (npr. izvorišna i odredišna IP adresa) i pomoću njih se izgrađuju meta-podatci koji se pripajaju na kraj paketa i šalju na izlazno sučelje.

Postoje dva tipa memorijskih modula koji se koriste u implementaciji: Block Random Access Memory (BRAM) i Quad Data Rate Static Random Access Memory (QDR SRAM). BRAM je memorija integrirana na FPGA pločicu, ograničenog je kapaciteta i veoma niske latencije (potrebna su do dva ciklusa takta za čitanje iz nje) a QDR je vanjski modul memorije većeg kapaciteta ali i nešto veće latencije (do oko 20 ciklusa takta za čitanje iz nje). Oba tipa memorije prikladni su za rad s velikim brzinama, pa se zato i koriste za filtriranje paketa: konkretno, u njih se spremaju podaci potrebni za izvođenje LPM algoritma koje sklopovlje šalje programskom filtru u meta-podatcima.

U predloženom sustavu koristi se programski filtar RFPF (engl. *Restricted Feature-set Packet Filter*), razvijen u prethodnom istraživanju. RFPF je filtar IPv4 prometa visokih performansi, s kojim se pokazalo da može filtrirati DDoS promet pri 10G brzinama korištenjem samo jedne jezgre CPU-a. Radi tako da se pomoću *netmap* programskog okvira veže za dva mrežna sučelja te iz zadanog popisa pravila generira kod u programskom jeziku C. Taj kod se pretvara u dinamički izvodiv program koji se "ubacuje" između mrežnih sučelja i tako filtrira promet u jednom i drugom smjeru. U ovom radu je prilagođen je hibridnom načinu rada tako da pri stvaranju C koda uzima u obzir način na koji se filtriranje odrađuje na sklopovlju te u programskom filtriranju koristi informacije iz meta-podataka. Programski filtar odvaja meta-podatke koji stižu sa sklopovlja od paketa s kojima dolaze i koristi ih u daljnjoj obradi.

Budući da su paketi koji dolaze do programskog filtra u implementaciji ovakvog filtra neovisno pripremljeni, tj. njihovi meta-podaci su stvoreni prije dolaska na njegovo ulazno sučelje, programskom filtru nije bitno što ih je stvorilo i kako su nastali. Za potrebe testiranja i validacije sustava odlučeno je pripremiti se za mjerenja tako da nema potrebe dizajnirati više različitih sklopovskih implementacija, već je programski "simulirano" prethodno filtriranje i sklopovsko stvaranje meta-podataka. To se radilo na odvojenom računalu, gdje su se paketi generirali i pritom automatski stvarali meta-podaci, dodavali na pakete i slali prema programskom filtru. Za generiranje paketa korišten je alat *pkt-gen* dodatno modificiran da po potrebi stvara i paketima dodaje meta-podatke.

Poglavlje 5 — Mjerenja i rezultati

U ovom poglavlju prikazana je usporedba rezultata filtriranja bez rasterećivanja na sklopovlju i s različitim rasterećivanjima na sklopovlju. Mjerili su se prosječna ukupna propusnost i prosječni broj potrošenih CPU ciklusa po paketu, a izvedena su dva tipa mjerenja: simulirana i hibridna. I jedan i drugi tip koriste generator prometa gdje se stvara promet s potpuno nasumičnim izvorišnim IP adresama te promet sličan DDoS napadu, s velikim brojem nasumičnih izvorišnih IP adresa ali koji pogađaju pravila iz liste pravila koja se u tom trenutku mjeri — da se pokaže kako filtar radi pod pritiskom. Za simulirani tip izmjereno je više različitih popisa pravila s različitim meta-podacima za rasterećivanje na sklopovlju gdje se za neke kombinacije dobilo i unaprijeđenje rada filtra i za 30%. Dodatno, potvrđena je i pretpostavka da je filtriranje uz LPM algoritam višestruko bolje od filtriranja bez njega.

Mjerenja su podijeljena po listama pravila, od kojih se za svaku listu testiraju meta-podatci povezani s tipom pravila u toj listi. Mjerenja za svaku pojedinu listu pravila izvršeno je višestruko da se dobije prosjek rezultata za sve tipove rasterećivanja: bez ikakvog rasterećivanja na sklopovlju (samo programski filtar bez meta-podataka), a zatim s promijenjenim parametrima za rasterećivanje na sklopovlju (npr. različiti broj pravila koja se na jednak način odrađuju u sklopovlju).

Da bi se potvrdila simulirana mjerenja, neka od mjerenja izvela su se i prikazala na pravom hibridnom sustavu, koristeći FPGA sklopovlje za stvaranje meta-podataka. Rezultati tih mjerenja potvrdili su rezultate istovjetnih simuliranih.

Poglavlje 6 — Zaključak

Kao zaključak, navedeni su znanstveni doprinosi ove disertacije: model podatkovne staze brzog klasifikatora mrežnog prometa temeljen na hibridnoj sklopovsko / programskoj kombinaciji FPGA s programskom podrškom na računalima opće namjene, model FPGA podatkovne staze rekonfigurabilne u približno stvarnom vremenu za potporu klasifikaciji mrežnih paketa u hibridnom sklopovsko / programskom filtru mrežnih paketa za vrijeme izvođenja, heuristička metoda raspodjele posla na sklopovsku i programsku komponentu za optimiranje propusnosti u podatkovnoj stazi hibridnog sklopovsko / programskog filtra mrežnih paketa, te metodologija empirijske evaluacije raspodjele poslova na sklopovsku i programsku komponentu u hibridnoj podatkovnoj stazi za filtriranje u mrežama visoke propusnosti.

Dodatno, valja naglasiti i skalabilnost ovakvog sustava, jer pri spretnom korištenju LPM algoritma za pretragu IP adresa, filtriranje ne ovisi o broju pravila, već o metodi rasterećivanja filtriranja na sklopovlju. Uz prikladno sklopovlje može se očekivati da će poboljšanje takvog sustava biti održano i pri većim brzinama. U međuvremenu je izašla nova NetFPGA PLUS razvojna pločica s brzinama prijenosa 100 Gbit/s, pa bi za buduće istraživanje bilo poželjno na njoj isprobati ovakvu vrstu hibridnog pristupa filtriranja i potvrditi njegovu skalabilnost.

Osim toga, u daljnjem istraživanju fokus će se prebaciti na isprobavanje nekih od ostalih postojećih LPM algoritama da se utvrde njihove prednosti i nedostatci pri njihovom rasterećivanju na sklopovlju ali će se testovi pokušati odviti i unutar prave mreže koristeći pravi promet.

Contents

Chapter 1

Introduction

1.1 Background and motivation

With the growth of the Internet and its rapidly increasing network speeds, there is a simultaneous increase in the need for high-performance and high-throughput network devices. In order to function properly and to provide the required services, these devices must be able to process the network traffic at high speeds, reaching tens or even hundreds of millions of packets per second. The lack of low-cost, flexible, and readily available high-throughput network devices that can meet these requirements is emerging as a problem, especially when it comes to packet classification and filtering.

The number of Internet users (both human and non-human) is rapidly increasing, and so is the need for fast networks and fast, configurable and easily-accessible network processing equipment. Thanks to continuous technological advances in recent years, high-speed networks became available to a large number of Internet users, and inexpensive and consumer-friendly networking equipment are available to everyone.

Any volume of data passing through an Internet Service Provider (ISP) or a datacenter at any given time must be available to their users, without interrupting their connections or dropping packets. This means that all network equipment must meet the same standard so that old or inadequate devices can be easily replaced or upgraded to avoid constant and expensive changes to the network infrastructure, while having the scalability of the equipment in mind. These network devices may be routers, firewalls, load-balancers, and other function-specific appliances used for various high-speed environments (e.g., ISPs, datacenters, etc.). Each type of traffic processing requires at least the header of each packet to be be examined individually. In order to keep up with the bandwidth of the network, all devices must be able to process all packets (or their headers) with adequate speed. Thus, a router in the ISP infrastructure should be able to handle any traffic that its link can support, for any frame size (from minimum to maximum).

Since the size of the packet headers is usually constant (and small), the total size of the packet depends mainly on the size of the payload. Packet processing generally involves checking the headers, so the total number of packets that can be processed depends solely on the total number of headers. This means that a fully utilized link will have the largest number of headers that should be analyzed and processed if all packets have a minimum size. Processing this many packets is supported by specialized high-end hardware, but for normal, commodity hardware such as PCs or servers, this limit is even more difficult to achieve — there must be enough time for the CPU to process each packet. For Ethernet, the minimum frame size (S) including preamble, start frame delimiter, and interpacket gap is 84 bytes and the maximum frame size (excluding jumbo frames) is 1542 bytes. If only minimum size packets traverse a link with a bandwidth of 100 Gbit/s (B), the total throughput (T) of this link can be calculated using Equation 1.1.

$$T = \frac{B}{S} = \frac{100 \cdot 10^9 b/s}{84 \cdot 8 b/pkt} = 148.81\ Mpkt/s \tag{1.1}$$

Using the CPU with $f = 4$ GHz clock frequency, this means that each packet (i.e., its header) must be processed in less than 27 cycles, as shown in Equation 1.2.

$$\frac{f}{T} = \frac{4 \cdot 10^9\ c/s}{148.81 \cdot 10^6\ pkt/s} = 26.87\ c/pkt \tag{1.2}$$

Comparing this with the throughput for maximum size packets, the maximum throughput is calculated in Equation 1.3 and the number of cycles per packet is calculated in Equation 1.4.

$$T = \frac{B}{S} = \frac{100 \cdot 10^9\ b/s}{1542 \cdot 8\ b/pkt} = 8.11\ Mpkt/s \tag{1.3}$$

$$\frac{f}{T} = \frac{4 \cdot 10^9\ c/s}{8.11 \cdot 10^6\ pkt/s} = 493.22\ c/pkt \tag{1.4}$$

This shows that the load on CPU is 18 times higher when minimum size packets are processed, compared to when maximum size packets are processed.

Different network elements perform different tasks. Each device defines what exactly to do with a packet once it reaches it. Network switches check the VLAN tags and / or destination MAC addresses of incoming packets and forward them to different destinations depending on their ARP tables or some other programming. Routers check the destination IP addresses and compare them to their internal routing tables so that they know to which interface and next-hop the incoming packets should be forwarded. Firewalls usually check different types of header fields of a packet to classify it and perform an action (drop, forward, etc.). Classifying packets in less than 30 cycles can be a difficult task, as multiple computations and memory fetches may be required. It is unrealistic to expect regular traffic to consist only of packets of minimal size, but nevertheless, each network element must be able to handle this volume of data. If even one element of the network infrastructure fails, the availability of the entire infrastructure is no longer guaranteed.

The need for high-speed packet processing is augmented by another factor: malicious traffic from various attacks on regular users and the network infrastructure, which can be found in every corner of the Internet and must be filtered out, making packet classification even more challenging. The author's earlier work in [1] explores the use of various attacks on hosts and networks and shows how easy it is to cause disruption to regular Internet users. It also shows that the approach to security assessment and research of these attacks can be done with commodity hardware and emulation software. This initiated research on how to use similar tools to protect against such attacks.

The malicious users use various methods to cause damage or disruption and steal data / money from their victims. One of the attacks that allows the attacker to cripple a network is the DoS (Denial-of-Service) attack. This attack requires either an exploitable vulnerability in a protocol / application used by the victim's service or a large volume of traffic to throttle the victim's link, leaving no bandwidth for regular users and denying them the service. Since widely used services require a link with sufficient bandwidth to support a large number of users, a relatively small number of attackers attacking the service must also have the same (or larger) bandwidth to keep the link busy. This is rarely the case, and this is where DDoS (Distributed Denial-of-Service) comes in: attackers "enroll" a large number of compromised hosts (PCs or IoT devices), called *bots*, to do the work for them (with or without their knowledge). Each *bot* can generate a relatively small volume of data and send it to the victim, but the combination of data from thousands (and even millions) of *bots* can easily flood the entire link and make the service inaccessible to regular users.

According to Google [2] and Cisco [3], the number of DDoS attacks is increasing exponentially and will reach about 15 million per year in 2023. There are different types of DDoS attacks, as described in this [4] taxonomy, but the main categories are: (i) volume-based attacks, (ii) protocol-based attacks, and (iii) application-layer attacks. The (ii) and (iii) differ from (i) in that the impact of these attacks can be mitigated by patching or repairing the vulnerable protocol or application. However, it is impossible to combat against sheer volume of traffic without distinguishing the 'good' packets from the 'bad' ones and somehow filtering out the bad ones before they reach the service. As mentioned earlier, this is particularly problematic when dealing with very high network speeds (10 Gbit/s and above).

Although it was planned that the IPv4 address space would soon be replaced by IPv6 due to recent exhaustion of IPv4 addresses, there is currently no indication that this will happen anytime soon. Although data from Google [5] suggests that more than 30% of users access its servers via IPv6, other sources, such as the Amsterdam Internet Exchange Service [6], show a much lower percentage of total IPv6 traffic, around 5%. A more realistic picture is provided by Akamai [7], where IPv6 traffic accounts for about 20% of all Internet traffic. Statistically, with a lower percentage of total IPv6 traffic, there are also fewer IPv6 DDoS attacks. So, since IPv4 is here to stay for a while, this thesis focuses exclusively on IPv4 traffic and combating IPv4 DDoS attacks, but the methods described can be used (with modifications) for IPv6 as well.

1.2 Thesis overview

The thesis is organized as follows:

Chapter II extends the already explained background and motivation of the thesis with a specific problem in high-speed networks: problems of mitigating the volumetric DDoS attacks. It describes the current state-of-the-art research and solutions to these problems. The overview of their positive and negative sides is given. The chapter further discusses the possible improvements to existing solutions by using a hybrid hardware / software system that utilizes Longest Prefix Matching (LPM) to speed up IP address lookups and packet filtering.

Chapter III describes the proposed model of the hybrid hardware / software system for filtering traffic, both the theoretical one and the one actually used in the implementation, as well as the differences and the reasons why some decisions were made the way they were. It also categorizes the rules used in firewalls to better prepare the separation of hardware and software workload.

Chapter IV describes the implementation of the hybrid system; divided into hardware, software and the distributor parts. The hardware implementation is described in detail, as it was designed and developed (almost) from scratch, and the software part shows the adjustments that had to be made to the existing software filter. The distributor part shows an empirically designed model for selecting the optimal offload for the hybrid system. It also discusses how to evaluate such a system and explains how it can be simulated without actually implementing all the features into the hardware.

Chapter V shows simulations with different types of rulesets, offload types and traffic, and evaluates each result. The differences between the results of the hybrid filtering and those of the baseline filtering are shown, and the choice of rulesets and offload types used are discussed. In addition to the simulation results, the results of one type of offload that was implemented in hardware are also shown.

Chapter VI concludes the thesis with the contributions of this thesis and gives an overview of possible improvements for this implementation for future work.

1.3 Summary of contributions

Main contributions of this thesis include:

- A model of a high-throughput network packet filtering datapath based on a hybrid hardware / software implementation combining FPGA and software running on commodity hardware. The model is described in Chapter 3.
- A model of a configurable FPGA datapath for offloading and assistance in a hybrid hardware / software network packet filter. The model is described in Chapter 4.
- A heuristic method for balancing the distribution of components for optimum throughput in the hybrid hardware / software packet filtering datapath. The method is described in Chapter 4.
- Empirical evaluation of the proposed distribution method for separating hardware / software components in a hybrid packet filtering datapath for high-throughput traffic environments. The evaluation is shown in Chapter 5.

Chapter 2

DDoS protection

Currently, the most common methods of protecting the service from large-scale volumetric DDoS attacks are either delegating the traffic to third-party companies (DDoS Protection Services) that clean and return the traffic using a technique called “scrubbing”, or using hardware-based filtering in packet switches and specialized appliances at the edge of the network. There is also the option to combine these two solutions: specialized DDoS defense appliances on-site for smaller attacks and the option to divert traffic to scrubbing centers if the attack grows larger and the on-premise appliances cannot handle it, as seen in this [8] comparison.

There are certain industries (e.g., financial sector) that must adhere to rules and regulations related to information privacy and data secrecy. Therefore, any option that requires traffic to be diverted to a third-party runs the risk of mishandling confidential information, and hence security and privacy risks. The only way to avoid this is to use on-premise appliances.

These appliances are network elements that use dedicated hardware and / or introduce proprietary software solutions. This makes them difficult (or impossible) to upgrade or modify, dependent on their manufacturers / vendors, and most importantly, expensive to purchase or even maintain, often requiring annual licenses for products that implement outdated solutions (or purchasing a newer model of the same type of appliance to meet ever changing infrastructure needs).

Another negative aspect of hardware-specific appliances is their high energy consumption. Due to their high throughput, Ternary Content-Addressable Memories (TCAMs) are the key technology for storing rules on filtering-capable devices (e.g., switches), but they suffer from limited space and high power consumption [9, 10], problematic configuration and maintenance, and the complexity of the algorithms used to translate Access-Control List (ACL) rules to the hardware [10].

In an attempt to match the classification speeds of hardware TCAM solutions, the work of [11] demonstrates the possibility of implementing high-speed generic decision trie lookups using only Field Programmable Gate Array (FPGA) technology, similar to [12], which uses

FPGA for high-speed Longest Prefix Matching (LPM), or pseudo TCAM [13], which emulates the behavior of TCAM in FPGA. Performing LPM lookups exclusively in hardware is limited by the lack of available memory, so implementing multiple large LPM tables would be of questionable feasibility, but offloading even one table to hardware opens the door to some alternative approaches. One of these has been explored in this thesis.

For older, slower networks, the traditional software packet processing tools and frameworks available in general-purpose Operating Systems (OS) were sufficient to withstand the (then) large amount of traffic with fine-tuning and with a relatively low penalty on the host computer's resource pool, as shown by the example of mitigating volumetric DDoS attacks with iptables[14] in [15]. Software packet processing and even network datapath emulation on commodity hardware has been possible for slower networks, as the author's own work [16] from 2014 suggests, but as network cards become commodity hardware at 10, 25, 40, and 100 Gbit/s, general-purpose firewalls and the datapath of the typical OS network stack cannot keep up with today's packet processing speed requirements. Even IPset [17], an iptables extension that can filter large rulesets by storing the rules as hash tables or bitmaps, cannot withstand the high throughput.

In recent years, several packet processing software frameworks have emerged, promising high-performance, flexibility, and reliability: such as Netmap [18], DPDK [19], or eBPF / XDP [20, 21] (used in CloudFlare [22] and Facebook [23]). They have become popular and mature enough that they can fight shoulder to shoulder with hardware-based devices in packet processing and filtering. There are a number of research works (e.g., [24, 25, 26]) that use these frameworks for packet classification, analysis, and processing, but their solutions still suffer from insufficient speed compared to specialized hardware devices.

Software Defined Networking (SDN), which is becoming increasingly popular, can also be counted in this category, mostly using OpenFlow [27] as a backbone for communication between the control and data planes. It focuses on a more generalized view of network manipulation and packet classification, which means that it suffers from some architectural limitations [28] in combating large volumetric DDoS attacks. Some methods and approaches to mitigate DDoS attacks using SDN (such as [29]), as well as Content Delivery Networks (CDN [30]) have been documented in this [31] survey.

An alternative to hardware and software solutions is something that takes the best of both worlds and combines both methods as a hybrid hardware / software solution. This combination can provide an inexpensive, flexible solution that is easily updated and maintained, yet fast enough to handle a large number of packets by leveraging the speed of hardware devices and the flexibility of software solutions. Another advantage is ease of use and the ability to modify and upgrade the solution without being as expensive as hardware appliances on the market. There are three common ways to combine hardware and software: by using FPGA technology [32, 33, 34, 35], Graphic Processing Units (GPU) [36, 37, 38, 39, 40], or software-assisted Network Interface Controllers (smartNICs) [41, 42].

This survey [43] of fast packet processing solutions lists and explains the most popular solutions, and gives some negative aspects of the hardware components. FPGA development is slow and complex due to low-level programming in Hardware Description Language (HDL). Therefore, it is not suitable for fast-response solutions when used in such a way that most of the logic resides on the FPGA itself (e.g., MPFC [44], where filtering rules must be re-synthesised and transferred to the board each time they are changed, a method also used by HyPaFilter+ [35]). All GPU-assisted hybrid solutions suffer from high latency, as they need to enforce packet batching to utilize the GPU parallelism, while some of them introduce packet re-ordering that may be unacceptable in certain network environments. Moreover, these solutions are not compatible with all GPUs due to their non-standard APIs and libraries, and GPUs are not as energy efficient as FPGAs. SmartNICs [45] can be FPGA-based, ASIC-based, or System-On-Chip-based (SoC-based). They are designed to offload processing tasks from the CPU. The price / performance / flexibility ratio highly favors SoC cards as they are easy to program, but their capabilities are limited compared to FPGA-based ones.

To evaluate the DDoS filtering solutions, the type of traffic to test them must be taken into consideration. Each solution needs to be exposed to the type and volume of traffic equivalent to that of a real DDoS attack. It is possible to simulate the attack by flooding the filter with synthetic traffic with randomly generated IP addresses or by using existing traces of DDoS attacks. CloudFlare [22] and Facebook [23] use genuine DDoS traffic to test their solutions because they have access to a large quantity of real-world data and various DDoS attacks. Most researchers do not have access to such data, so solutions such as [26, 34, 35, 41, 46] use synthetic traffic to simulate DDoS attacks in their tests. By using a large pool of randomly generated IP addresses, traffic generated in this way can approximate DDoS attacks. With no more than tens of thousands of IP addresses used in some of the aforementioned works, such synthetic traffic cannot compare to some of today's volumetric DDoS attacks. For example, the attack on Dyn in 2016 had tens of millions of different IP addresses according to their report [47] (Dyn's original report is no longer available, but there are still various analyzes of it on the Internet [48, 49, 50]). Any DDoS filtering system must be able to withstand such attacks.

The most effective approach to ensure that each incoming packet is processed is to filter packets sequentially (one-by-one) by matching them against defined source / destination IP addresses and ports, and other specific patterns in incoming packets. However, filtering volumetric DDoS attacks can be accomplished much better and easier by placing specific source / destination IPv4 addresses in an allowlist or a blocklist. For appliances in larger networks, maintaining relatively large databases with allowlisted or blocklisted IP addresses or IP ranges and searching those databases would yield better results than writing a large number of individual rules for each IP address and processing those rules on a per rule basis. For this type of search, LPM would have to be utilized, which hardly any of the above mentioned solutions even take into consideration.

Besides the “active” protection measures against DDoS attacks, there is also the possibility to protect the attacked network infrastructure by blackhole routing / filtering (blackholing). This involves contacting the upstream providers with the target IP addresses so that they can stop routing all incoming traffic and save bandwidth. The result is that the victim is unreachable to all users — both the attackers and everyone else, which essentially means that the DDoS attack was successful. This method is used as a last (and often only) resort when there are no other ways to protect the network other than shutting down the target and protecting the rest of the infrastructure.

It is important to distinguish between DDoS protection (mitigation) and DDoS detection (recognition). Although DDoS mitigation systems may include DDoS detection, this is not always the case. The focus of this thesis is on filtering packets and mitigating DDoS attacks that have already been detected, assuming that some other system(s) perform the task of automatic or non-automatic DDoS detection.

2.1 FPGA

There is a plethora of projects in various research areas that utilize FPGA as the core of their system, so it was considered the biggest candidate for the hardware part of the system in this thesis. For example, one of the early trials was performed by Microsoft [51], where FPGA hardware was demonstrated as a viable and efficient accelerator for web search.

There are also various efforts to use FPGA parallelism as a method for hardware or hybrid network defense, such as a theoretical firewall shown in [52], but lacking additional functionalities for a real hybrid system (it demonstrates only packet header parsing) or [53], another simple but reconfigurable FPGA firewall that does not achieve 10G speeds. Other examples include Network-based Intrusion Detection Systems (NIDS) such as [54], an older reconfigurable hybrid system where the pre-filtering hardware offloads parts of *Snort*[55] rules to improve its work, a purely hardware, FPGA-based reconfigurable NIDS [56], or a high-level language (PP) for describing packet parsing algorithms for FPGA-based packet parsers [57]. The key-value store LaKe [58] uses a similar hybrid model for its FPGA-accelerated hybrid *Memcached* [59] architecture. Although it does not address DDoS attacks or their mitigation, it demonstrates a similar model of a hybrid hardware / software system for exchanging information between FPGA and CPU as the one proposed in this thesis.

2.2 Packet classification

Packet classification is mostly used in the network by Packet Classification Engines (PCE) to classify packets according to some rules in the ruleset and to assign them a flow identifier (*flowID*). Different packets can then be assigned the same *flowID* based on some criteria (standard 5-tuple or more fields) and the same *flowID* can be matched later to make some decisions for any kind of network application, including traffic filtering.

According to this taxonomy [60], packet classification can be divided into four different techniques: exhaustive search, decision tree, decomposition, and tuple space. All four techniques start from the same basic assumption: There are a large number of rules with different fields that need to be classified according to certain criteria. Each technique, with the exception of exhaustive search, focuses on the development of methods and algorithms that optimize filtering by minimizing the size and complexity of the ruleset used.

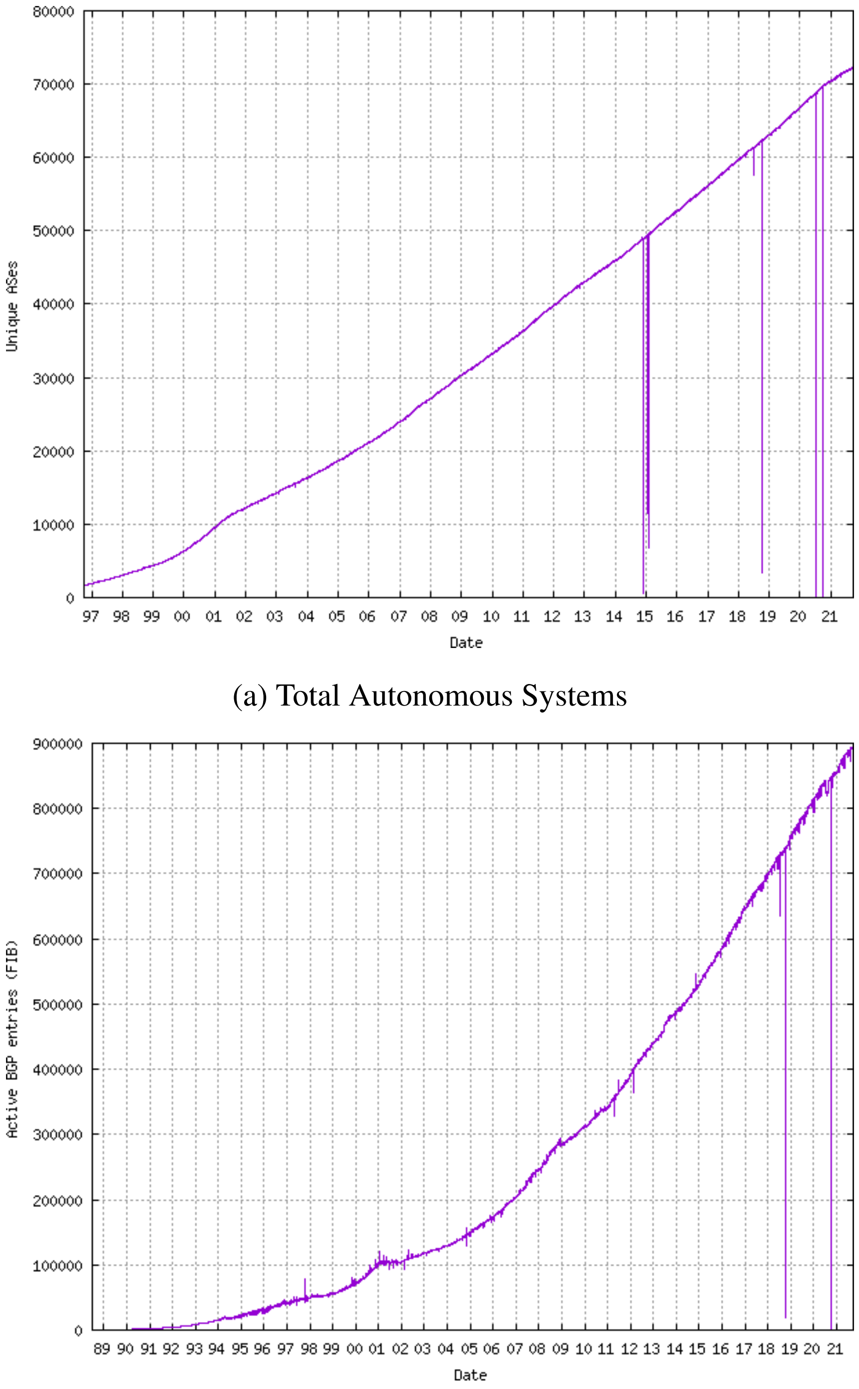
For example, there are decision tree algorithms [61, 62, 63] that treat rulesets as multidimensional spaces that can be “cut” depending on the overlap of these spaces, thus removing branches from their decision tree. These algorithms are often complex to design and develop and may even take hours to preprocess, depending on the size of the ruleset (although there are modern “cutter” algorithms that can do this in seconds for the same rulesets, such as CutSplit [63]), and even then their performance depends on the implementation and often fails to achieve speeds of TCAM hardware.

To test their methods, PCEs most often use the ClassBench [64] rule generator, a de-facto standard for creating synthetic but realistic rulesets based on predefined statistical data.

2.3 Improving packet filtering

With the prevailing increase in the number of Internet users and the number of Autonomous Systems (AS), the total number of prefixes in the global IPv4 Border Gateway Protocol (BGP) routing table is also increasing, as shown in Figure 2.1. For this reason, the search for fast, flexible, and cost-effective solutions for routing traffic has led to advances in IP lookup in software and the development of new LPM algorithms ([65, 66, 67]), as opposed to using dedicated hardware.

The work on software routing from [69] provided the impetus to further pursue LPM as a possible method for high-speed packet filtering. It promised great potential, as shown in the work of the author of this thesis [70], but still left room for improvement. The original idea to improve such software packet filtering was to create a hybrid hardware / software system in which packets are processed and filtered either partially by offloading some parts to the reconfigurable hardware of the Network Interface Card (NIC) or completely in the NIC. In the



(a) Total Autonomous Systems

(b) Total BGP table size

Figure 2.1: The increase of the Autonomous Systems and the growth of the IPv4 BGP routing table as of 09/2021. Source: BGP Routing Table Analysis Reports [68].

case of partial offloading, the packet must be processed by both the hardware and the software. This means that the hardware performs some necessary processing of the packet and forwards it to the software along with some additional metadata to help the software to speed up its own processing. Complete offloading, where filtering is done entirely in hardware, saves software resources and speeds up the filtering process compared to filtering only in software.

This thesis focuses on reducing the total number of rules in the ruleset by using LPM while maximizing the total number of IP prefixes covered by the filter and minimizing the reconfiguration time when the firewall ruleset configuration needs changing.

In the algorithms used in previous work, memory fetches proved to be the bottleneck in this type of software filtering. To improve filtering throughput and reduce the load on CPU, parts of the LPM algorithm could be computed in hardware, which would then forward the required data to software. In addition to offloading the LPM algorithm, other types of filtering rules could also have been offloaded through the careful elaboration and crafting of metadata communicated between the hardware and the software. The thesis also shows research done in this area.

The aforementioned PCEs are mostly intended to be used by hardware (ASIC, FPGA, TCAM) due to their optimizations for parallelization. The assumption of filtering traffic with myriad of rules and multiple matching fields, while good for working with flows, makes little to no difference against a large volumetric DDoS attack, since IP addresses are mostly random (source addresses are sometimes spoofed) and flows are virtually non-existent. Regardless, PCEs should not be completely dismissed as they can be useful in future research by integrating some of the existing schemes (such as StrideBV [71] and WeeBV [72]) as they have presented an FPGA-based PCE capable for high-speed filtering systems with fast reconfigurability, suitable for DDoS attacks response.

Chapter 3

Hybrid system model

The hardware part of the hybrid system is built using the NetFPGA SUME board. NetFPGA boards are used for prototyping, research and development of high-speed networking systems. Since the early 2000s, NetFPGA boards have evolved from 1G (NetFPGA 1G [73, 74], NetFPGA CML [75]) and 10G Ethernet interfaces (NetFPGA 10G, NetFPGA SUME [76]) to the latest model brandishing 100G dual-port interfaces (NetFPGA PLUS). During this time, their core elements were also improved by using better and faster FPGAs. At the time of starting this thesis, the latest model was NetFPGA SUME, which is used in a variety of projects and research ([77, 78, 79, 80], etc.), even now, more than six years after its introduction. It uses Xilinx Virtex-7 690T FPGA, and has four 10GbE enhanced small form-factor pluggable transceivers (SFP+), three x36 72Mbits Quad Data Rate (QDR) II SRAM memory modules, two 4GB DDR3 SODIMM memory modules and other peripherals.

NetFPGA SUME can be used either as a separate hardware network device with the complete datapath designed in the FPGA, or as a NIC connected to a host. The idea for this hybrid prototype is based on the assumption that NetFPGA is configured as a NIC. Such configuration would make this hardware / software hybrid a standalone network middleware element installed at the edge of the network to be protected, using separate datapaths for incoming and outgoing traffic. The model for such a packet filtering prototype is shown in Figure 3.1. Since NetFPGA SUME has four network interfaces, this means that it is possible to create two separate datapaths using two pairs of interfaces ([nf0]–[nf1] and [nf2]–[nf3]) or to bundle both datapaths into one with double bandwidth ([nf0+nf1]–[nf2+nf3]). The packets arrive at one of the ingress interfaces of the NetFPGA and are then forwarded to the internal FPGA logic for offload, which parses the packet headers and, if necessary, creates and appends the metadata to the packet. If no additional filtering is required (i.e., everything has been decided in hardware), the packet is either discarded (dropped) or forwarded to one of the egress interfaces. Otherwise, it is sent over the PCIe bus to the software part of the filter, along with the metadata. The software then parses the packet and drops it or removes the metadata and forwards it back to the hardware, which sends it to the appropriate egress interface. Viewed from the other connected network devices, the packet enters the filter and leaves it unchanged.

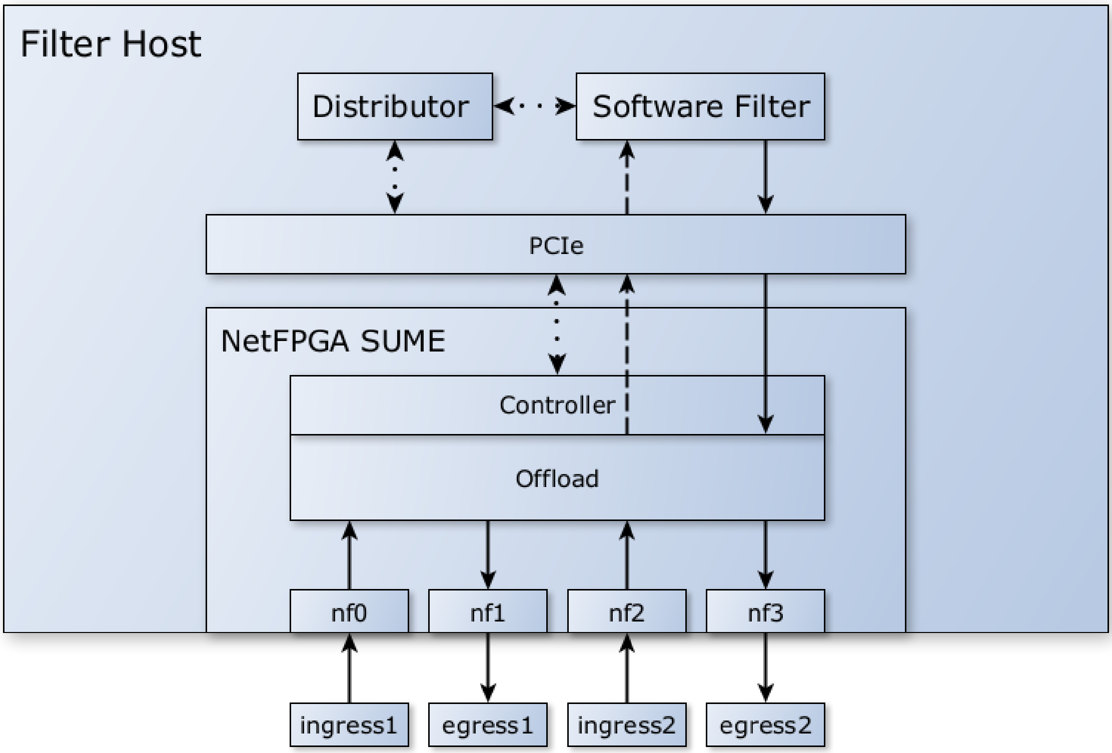


Figure 3.1: Architecture of the proposed DDoS filtering system using NetFPGA SUME NIC to combine hardware and software filtering. Different arrows represent different data being transmitted: regular arrows are packet datapaths, dashed arrows are packet and metadata datapaths, dotted arrows are communication between different modules of the system.

The software part of the system can be implemented using any kind of software filter (firewall) that can access the packet structure and modify it if necessary. If metadata is attached, the filter must truncate it before forwarding it to the egress interface. For this thesis, a modified version of an existing stateless filter described in previous research [70] was used. This filter was chosen because the author is familiar with it, its overall performance is better compared to similar tools, and its existing features are compatible with the assumptions of this thesis. Even as a software-only firewall, it provided high throughput with low resource utilization, but hardware offloading could provide additional improvements to further reduce packet processing time.

The distributor is an intermediary for communication between the user and the system as a whole, specifically the hardware controller and the software filter. It is a model used to determine how the packet is parsed in hardware, what metadata is created, and what filtering rules are appropriate for hardware offloading. The distributor must take into consideration multiple parameters before making any decisions: types of rules in the ruleset and its total size, hardware capabilities (e.g., types of packet parsers implemented, amount of memory), software capabilities (e.g., types of implemented LPM algorithms, limitations on the number of rules), current network status (e.g., currently under a DDoS attack), and type and volume of traffic arriving at the filter.

As already mentioned, to implement this design, NetFPGA SUME should be used as a NIC connected directly to the host via the PCIe bus. By using the available design (Reference NIC project from the NetFPGA SUME repository[81], based on the RIFFA[82] generic FPGA DMA engine), the NIC could not achieve the desired throughput for high-speed packet filtering. Since the driver included in the project did not support the *netmap* [18] framework, it was decided to create a new, improved driver for the FreeBSD Operating System (OS) that would hopefully achieve better throughput.

During the development of the new driver, some obstacles were encountered that could not have been avoided. In order to take advantage of the *netmap* framework support in FreeBSD OS, the driver had to be developed using the *iflib* [83] framework. The NetFPGA SUME DMA engine, which controls the communication between hardware and software, uses only one connection to the host OS for all four physical SUME interfaces. This violates the operation of the *iflib* framework, which requires one connection for *each* physical interface. For this reason, the *iflib* method of developing the driver was abandoned in the early stages of development. This meant that the driver had to be developed without the *iflib* framework and had to rely on code from the existing Linux driver.

Further research and work on the driver revealed that the DMA engine used in the NetFPGA SUME Reference NIC project was not implemented in such a way that it could fully utilize the PCIe bus for data transfer between the NetFPGA SUME card and the OS. In addition, the existing Linux driver had bugs that further slowed down their communication, and a hardware design bug was found that caused the NIC to stop working at some point when transmitting traffic. Due to this bug, the NetFPGA SUME board had to be reset to start working again.

Aside from the bugs fixed, the newly developed driver has been found to work slightly better than the existing one, with the more balanced receiving and transmitting of TCP traffic, the watchdog for the existing hardware transmit bug, working link status on the host, and hardware traffic counters. However, due to the aforementioned limitations in the hardware implementation of the DMA engine, it was still not fast enough to work with 10G networks, so it is not suitable for this type of filtering. The alternative was to use a different DMA engine.

There were two new DMA engines and their accompaniying drivers developed by the NetFPGA community: NAUDIT DMA [84] with the UAM driver and the Corundum [85] DMA engine / driver combination. Neither of them could achieve line-rate for NetFPGA-to-host communication for minimal sized network packets. Usage of the NAUDIT DMA engine and corresponding driver lacked the speed to transfer small packets over PCIe. According to the referenced paper, the 10 Gbit/s throughput would only be possible for device-to-host transfers of at least 2 kilobytes of data (host-to-device transfers require even larger payloads). This would be inefficient unless the packets were larger or transferred in batches, but further research has shown that the accompanying driver could only process about 700 Kpps for 64-byte packets and did not support *netmap* framework. The Corundum NIC suffered from the same problems.

For this reason, and because of the complexity of developing a new DMA engine, it was decided to change the system model so that the PCIe bus between the SUME card and the software was not used for packet transfer, but only for minimal communication between the distributor and the controller on the FPGA.

It was decided that in the new model, metadata should be forwarded from the FPGA to the software filter via Ethernet. Even though this type of communication affects the overall throughput of the entire system, it achieves sufficiently high speeds so that the system can be used in 10G networks. The new model is shown in Figure 3.2.

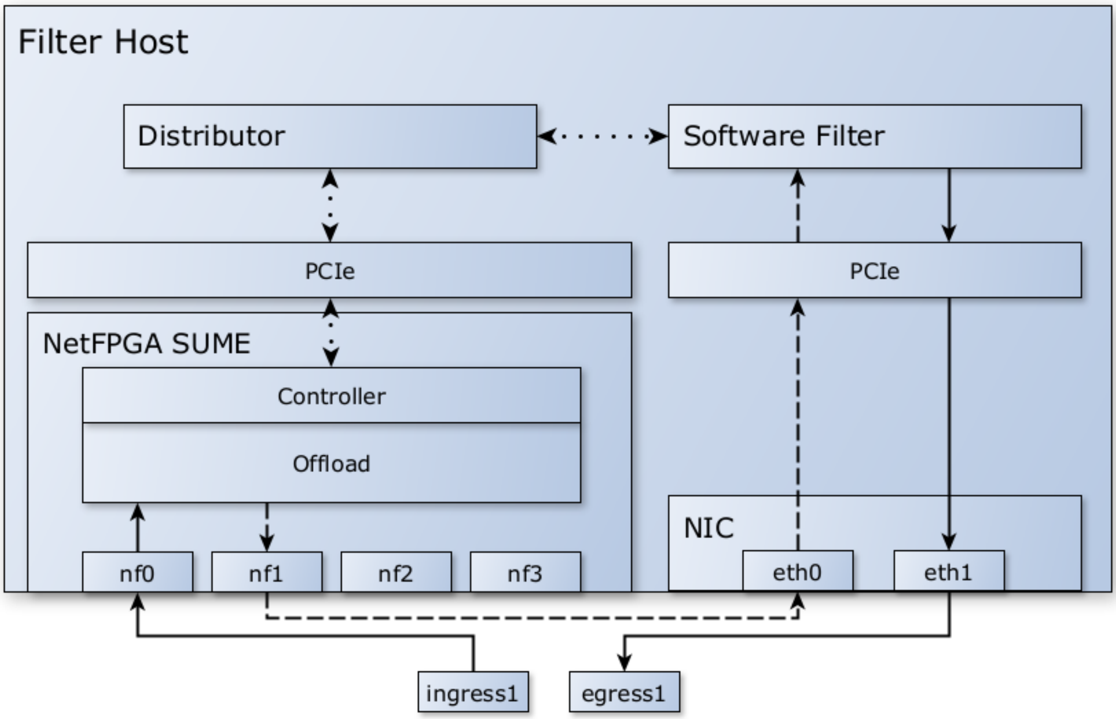


Figure 3.2: Architecture of the implemented DDoS filtering system using NetFPGA SUME to combine hardware with software filtering without using the SUME NIC design to communicate with the software filter. Different arrows represent different data being transmitted: regular arrows are packet datapaths, dashed arrows are packet and metadata datapaths, dotted arrows are communication between different modules of the system.

This model is similar to the previous one, but uses an additional NIC that receives packets along with the metadata and forwards them to the software filter. Before the packet reaches the software, the NetFPGA performs the same task as in the previous model, except that it is not able to automatically forward the packets to the egress interface, so this type of offloading is disabled. To simplify the implementation, only one NetFPGA SUME interface pair is enabled ([nf0]–[nf1]) for this model, since another one would require an additional NIC on the software side. However, the model can be extended with another pair to increase the overall throughput of the entire prototype.

3.1 Rule categorization

Before the system starts processing packets with the given ruleset (list of rules), the rules must be categorized according to where they will match the packets: in hardware, software, or both. This is done internally by the distributor component of the system after the rules are parsed, so the user does not have to be concerned about the rule categorization process. Each category is determined by how it can be *offloaded* to hardware in this implementation: fully, partially, or not at all.

The prototype software filter used in this paper is based on the stateless filter described in [70]. It uses a similar rule definition syntax as other standard firewalls. The filter's ruleset consists of rules that are applied to each packet in turn (sequentially). When a rule is found that matches the packet, the corresponding action is performed.

Each rule has an *action* associated with one or more *patterns* (implicitly linked by a logical operator AND), in the format:

action pattern {pattern . . . }

To determine whether individual rules can be completely offloaded to hardware, partially offloaded to hardware or not offloaded at all, they are categorized by combining *action* attributes with *patterns* attributes. The *action* and *patterns* attributes of the rule are defined as follows:

- *action* attributes
  - termination — does the rule terminate, i.e. should the filter stop processing the rules after the currently matched one,
  - counting — does the rule use software counters, i.e. should the counter for that rule be incremented if the packet is matched.
- *patterns* (rule *body*) attribute
  - offload — can the combination of the rule *patterns* be processed in hardware and how (fully or partially).

3.1.1 *Termination* attribute

Rule *actions* can be: ACCEPT, NC_ACCEPT, DENY, NC_DENY, or COUNT. ACCEPT and DENY rules forward or drop packets, respectively, but at the same time they signal the software to increment the counter for the associated rule. NC_ACCEPT and NC_DENY perform the same actions but have no associated counters. COUNT rules are used only to signal the software to increment the appropriate counter. The *termination* attribute can have one of three possible values:

- Accept termination (*A*) — *actions* that forward packets (NC_ACCEPT and ACCEPT),
- Deny termination (*D*) — *actions* that drop packets (NC_DENY and DENY),
- No termination (*N*) — *actions* that only count packets (COUNT).

Rule *termination* can affect its ability to be offloaded — e.g., if the packet containing an NC_ACCEPT rule matched on the hardware, it must be forwarded to software with the information that no further classification is required. The software should forward this packet to its egress interface without further processing. In this thesis, packets are assumed to pass through the hardware and software "serially", i.e., first the hardware, then the software. If the structure of the hybrid model changes, this assumption may change for NC_ACCEPT rules as well — the hardware could forward the packet without notifying the software. Whether or not the software is notified also depends on the position of the *terminating* rule within the ruleset. If there are non-offloaded *terminating* rules before the offloaded rule, the hardware cannot perform an *action* because of the possibility of non-offloaded rules matching before the offloaded one.

3.1.2 *Counting* attribute

Counting attribute can have one of two possible values:

- Count (*c*) — counters are incremented for these rules (*actions* ACCEPT, DENY and COUNT),
- do Not count (*n*) — there are no counters associated with these rules (*actions* NC_ACCEPT, NC_DENY).

Counting can provide packet classification statistics and, together with an external analysis tool, form a complete DDoS protection system with automatic detection of suspicious traffic behavior. Some rules may have associated counters that would be sent to external software for further analysis. As with *terminating* rules, the implementation of *counting* rules also depends on the model of the hybrid system. In this thesis, all packets first pass through the hardware while the software manages the counters. For this reason, when matching the rules that need to be counted, the hardware must notify the software and those rules cannot be fully offloaded.

Since the implementation of a hybrid filter could be used in conjunction with a separate DDoS detection system, rules with *non-counting*, *terminating actions* (NC_ALLOW and NC_DENY) do not make much sense. Offloading these rules to hardware and filtering them without any communication with the software would cause the system to lose potentially useful information about incoming traffic. These types of *actions* are not implemented in the prototype of this thesis, but are nevertheless included in this categorization for the sake of completeness.

3.1.3 *Offload* attribute

A *pattern* is a minimal expression in the rule that can be true or false, e.g. “destination port is equal to 80”. Depending on the implementation, each *pattern* can belong to one of three possible groups:

- Offloadable (p_O) — can be done completely in hardware. For example, comparing a packet protocol field with a value saved in internal hardware memory,
- Partially offloadable (p_P) — can be partially done in hardware. For example, an LPM algorithm that can be divided into multiple stages can have the first stage done in hardware,
- Non-offloadable (p_N) — cannot be done in hardware.

By combining multiple *patterns*, the rule *body* changes and so does the ability to offload rule processing from software to hardware. Depending on the type of *patterns* in the rule *body*, the *offload* attribute can take one of five possible values:

- fully Offloadable (O) — every *pattern* in the rule is offloadable (p_O). It is possible to offload all the *patterns* in the rule *body* to hardware,
- Partially offloadable, type 0 (P_0) — there are offloadable *patterns* (p_O), mixed with one or more non-offloadable *patterns* (p_N). All offloadable p_O *patterns* can be offloaded to hardware,
- Partially offloadable, type 1 (P_1) — there are partially offloadable *patterns* (p_P) mixed with zero or more non-offloadable *patterns* (p_N). Only parts of p_P *patterns* can be offloaded to hardware,
- Partially offloadable, type 2 (P_2) — there are partially offloadable *patterns* (p_P) mixed with zero or more non-offloadable *patterns* (p_N) and one or more offloadable *patterns* (p_O). Parts of p_P *patterns* and all offloadable p_O *patterns* can be offloaded to hardware,
- Non-offloadable (N) — every *pattern* in the rule is non-offloadable. It is not possible to offload anything to hardware.

Table 3.1 shows *offload* attributes defined by the combinations of the rule *patterns*.

Table 3.1: Rule *pattern* combination types. Types of *patterns* that repeat once or more are marked with $()^+$. Types of *patterns* that occur zero or more times are marked with $()^*$.

O	$(p_O)^+$
P_0	$(p_N)^+(p_O)^+$
P_1	$(p_P)^+(p_N)^*$
P_2	$(p_P)^+(p_N)^*(p_O)^+$
N	$(p_N)^+$

3.1.4 Offloading rules

Combining the values of the *termination*, *counting*, and *offload* attributes categorizes the rules according to whether and how they can be offloaded to hardware. There are a total of 30 possible combinations of all rule attributes (three possible *termination* options, two *counting* options, and five *offload* options). In the prototype used for this thesis, there are currently no *Non-terminating* rules that *do Not count*. Therefore, they are not shown, but the remaining 25 combinations can be seen in Table 3.2.

Table 3.2: Offload types for rule categories.

Hardware	ODn
Hybrid	OAc, OAn, ODc, ONc
	P_0Ac, P_0An, P_0Dc, P_0Dn, P_0Nc
	P_1Ac, P_1An, P_1Dc, P_1Dn, P_1Nc
	P_2Ac, P_2An, P_2Dc, P_2Dn, P_2Nc
Software	NAc, NAn, NDc, NDn, NNc

It is obvious that most of the rule categories are those that can be partially offloaded to hardware. The only category that can be fully offloaded is *ODn* — rules that consist *entirely* of *offloadable patterns* (p_O), with the *Deny termination* and *do Not count* attributes. There are cases where even these rules need to communicate with the software: when there are other rules before them in the ruleset that need to be checked by the software. There are five *Non-offloadable* rule types — any type of rule that consists *entirely* of *Non-offloadable patterns*.

The example of a pseudo ruleset with its categorized rules is shown in Figure 3.3.

As mentioned earlier, rules that are partially offloaded to hardware must pass information about the success of the classification operation to the software. To do this, the hardware must attach certain metadata to each forwarded packet so that the software can analyse it and act accordingly.

3.1.5 Metadata

Depending on the category of each rule, different information must be conveyed to the software. Rules that are processed entirely in hardware or in software do not require metadata, since no communication between hardware and software is required in these cases (with the exception when the order and the outcome of such rules affect the final outcome).

Since metadata fields vary in size, they need to be categorized and separated according to the rule type so that they can be later categorized based on priority. The transfer of metadata could be limited by bandwidth, so it is important to use the available data in an intelligent way.

$$\underbrace{\underbrace{NC\_DENY}_{Dn}\ \underbrace{\overbrace{(destination\ port == 22)}^{p_O}\ AND\ \overbrace{(protocol == TCP)}^{p_O}}_{O}}_{ODn}$$

$$\underbrace{\underbrace{ACCEPT}_{Ac}\ \underbrace{\overbrace{(destination\ port == 80)}^{p_O}\ OR\ \overbrace{(destination\ port == 443)}^{p_O}}_{O}}_{OAc}$$

$$\underbrace{\underbrace{COUNT}_{Nc}\ \underbrace{\overbrace{(source\ IP\ in\ table\ GOOD)}^{p_P}\ AND\ \overbrace{(destination\ port == 80)}^{p_O}}_{P_2}}_{P_2Nc}$$

Figure 3.3: Three types of rules, annotated with their categories. The first rule specifies that every TCP packet with destination port 22 should be dropped without counting. The second specifies that every packet with destination port 80 or 443 should be forwarded and counted. The third rule specifies that packets with the source IP address from the *GOOD* table and with destination port 80 should be counted.

If the *fully Offloadable* rule whose *termination* attribute is *Accept* or *Deny* ($O[A|D]*$) is matched, it is not necessary to check the subsequent rules in the ruleset. The software then needs to know that the rule is matched and perform the required *action*. In this case, the software only needs to know the ordinal number (ID) of the first *terminating* rule that matched in hardware. If the rule has the *Count* attribute ($O[A|D]c$), the software increments its counter if the rule matches, so no other metadata is required. As mentioned earlier, in the case of rules with the *do Not count* attribute, *ODn* rules at the beginning of the ruleset are the exception; they do not need to send any metadata to the software.

In the case of P_0 rules, which are a combination of non-offloadable (p_N) and fully-offloadable (p_O) *patterns*, the hardware cannot know the final result of the classification, only the results of the individual p_O *patterns* in the rule. This means that nothing but the result of each *pattern* of that type needs to be sent to the software: one bit for each p_O *pattern* (match or no-match).

P_1 rules are a combination of *partially offloadable* (p_P) and zero or more *non-offloadable* (p_N) *patterns*. The hardware independently computes some data useful for p_P *patterns* and forwards this data to the software. The size of this data is variable and depends on the *pattern* used.

P_2 rules can be seen as a combination of P_0 and P_1 rules, so the metadata needed for them is the same as the metadata for P_0 and P_1 rules combined.

It is possible to further compress Table 3.2 by having the character $*$ stand for one of the attributes it replaces, as seen in Table 3.3. This compression is possible because some categories share the same metadata fields. The third column shows the information that must be sent to the software along with the packet when a rule (or a *pattern*) is offloaded to hardware.

Table 3.3: Offload types for rule categories and metadata types — compressed view.

Hardware	ODn	no metadata
Hybrid	ONc	bits used for *Count* attributes
	OAc, ODc, OAn	ID of the matched rule
	$P_0 **$	a bit for each match / no-match p_O *pattern*
	$P_1 **$	parts of results / data for every p_P *pattern*
	$P_2 **$	combined P_0 and P_1 metadata
Software	$N **$	no metadata

3.1.6 Metadata field sizes

Types and sizes of metadata fields defined in Table 3.3 can be divided and described as follows:

- Partial offload — the size of metadata depends on the type of the offloaded element.
- Partial p_O *patterns* — one bit for each p_O *pattern* in a P_0 or P_2 rule.
- Rules with the *Count* attribute — one bit for each rule that is counted.
- Matched rule information — if the offloaded rule is matched, the software only requires its ID, i.e. the rule number. The size of this field depends on the total number of offloaded rules using this metadata, so it can represent any rule number in the ruleset — $\log 2(\mathit{total\_number\_of\_offloaded\_rules}) + 1$ bits.
- None matched — only one bit is needed, marking true or false.

Each field is rounded up to 1 byte (8 bits) to simplify the software part of the implementation.

Partial offload

When a *pattern* is *Partially offloaded*, it cannot be marked as matched or non-matched in hardware. Rather, the software filter uses the data sent by the hardware to determine whether or not the packet matches the *Partially offloaded pattern*. For example, if a software filter compares the UDP source port to a specific value, the filter must first parse the Ethernet, IP, and UDP headers of the packet and extract the required value. Instead, the hardware can "prepare" the UDP source port in the metadata so that the software can always extract it from the same location without parsing the packet headers. The hardware can also do partial offload if the software filter needs to perform LPM lookups based on IP addresses: first stages of the LPM algorithm can be performed on hardware so that the computed intermediate data can be forwarded to software. The software can then complete the LPM lookup using the received data.

Depending on the type of offloaded data and the implementation of the software filter, this data can vary in size and may contain different useful information.

Partial p_O *patterns*

For each partial *pattern* p_O, the hardware must determine whether the filter matches it and what is the result of this filtering: it can be 0 (no match) or 1 (match). Each incoming packet must be matched against each of these *patterns*, which are stored in the hardware's internal memory. This means that the amount of memory required for this metadata correlates with the number of p_O *patterns* in the ruleset. For example, if the ruleset contains X p_O *patterns*, where each of them checks whether the destination port is equal to a certain value, X bits must be stored in hardware for each of the possible destination ports of the incoming packets ($X \cdot 2^{16}$ in total). The larger X becomes, the more data must be stored in hardware and the more metadata must be sent to software.

Rules with the *Count* attribute

If there are *fully Offloadable* rules in the ruleset that need to be counted in the software, the information that the rule is or is not matched must be forwarded to the software. Assuming that the logic for matching these rules is implemented in hardware (e.g., as in the previous paragraph), no additional logic is required to extend this metadata. However, there is the same storage cost in hardware and overhead when transfering to software that grows with the number of such rules in the ruleset.

Matched rule information

When a rule from the "middle" of the ruleset is *fully Offloaded* to hardware, it is possible for the packet to also match any rule before it. Hardware has no information about the rules that are not offloaded, or about whether the packet matched any of those rules and what *action* is required. Therefore, it must not perform any *actions* on the packets that match rules from the "middle". The hardware must forward the information about which offloaded rule is matched as an exact rule number (ID). The software then has the final say on the fate of the packet.

None matched

The last column specifies an optional metadata field that can be used for rules with the *fully Offloadable* attribute. The *fully Offloadable* rules can use this field to allow the software to skip checking rules in certain circumstances. If they are the first rules in a ruleset, they can be skipped altogether without having to check them individually. In the implementation for this prototype, this bit can be included in any of the previous metadata so that a zero value means that nothing is matched. Therefore, this type of metadata is not separately evaluated.

3.2 Offload summary

Rulesets in firewalls are usually combinations of multiple rules, often belonging to different categories. It is necessary for the hardware to transmit multiple metadata fields at once to offload the work as intended. The size of the metadata information to be transmitted depends on the number of rules and their type. In some cases, due to limited bandwidth between hardware and software, not all metadata can be included in a single metadata transfer, so tradeoffs must be made. Although the conditions that must be met for offloading rules (or parts of them) have been described, the use of metadata mainly depends on the capabilities and capacities of the hardware and software. The final decision on offloading is made by the distributor based on its input parameters regarding both this categorization and hardware / software resources.

Table 3.4 shows the complete and detailed view of each rule category and its metadata, as explained previously. The *ODn* rules can be offloaded to hardware without sending any metadata to software only if they are at the beginning of the ruleset. Otherwise, the result of their check must be sent to the software by using the same metadata as the *OAn* rules. However, due to the previously explained architecture of the implementation from this thesis, the categories from the first two rows (*ODn* and *OAn*) are not used in the performance evaluation in the next chapters.

Table 3.4: Metadata fields for different rule categories.

	Partial offload	Partial p_O *patterns*	*Counting* rules	Match rule number	None matched
ODn					
OAn				**x**	**x**
ONc			**x**		**x**
$O[A\|D]c$				**x**	**x**
$P_0{**}$		**x**			
$P_1{**}$	**x**				
$P_2{**}$	**x**	**x**			
$N{**}$					

Chapter 4

Packet filtering implementation

To protect the network against unwanted, malicious traffic, it is necessary to monitor incoming and outgoing traffic in some way. It can be done either with automated tools or manually. Volumetric DDoS attacks can be noticed by an increased volume of traffic in the network infrastructure, especially when all traffic is directed at a small number of targets. To successfully protect against the most powerful DDoS attacks, i.e., those with the highest traffic volumes and the largest number of different source IP addresses, it is necessary to protect against tens of thousands or even millions of potential attackers that are using both real and spoofed source IP addresses.

To defend against such DDoS attacks, standard firewalls used by OS's are not sufficient. The paper [70] shows an alternative: using allowlists and blocklists that can classify much larger volumes of traffic by using the LPM algorithm to check IP addresses and / or network prefixes as they enter the network. Cleverly and quickly adding offending and other suspicious IP addresses to the list that blocks them will prevent such traffic from entering the network. It is also possible to have a pre-prepared list of secure and vetted IP addresses and / or network subnets that are forwarded to the network based on certain criteria (e.g., geolocation of IP source address — geoIP). This reduces the total number of rules (as these lists are stored and managed separately) and allows for easier firewall configuration and management.

There are various (academic and production) automated solutions [86, 87, 88, 89] for detecting such DDoS attacks, and complementing these solutions with the filtering system (firewall) described previously could lead to an effective defense against DDoS attacks with very low response times. In such cases, the communication between the detection system and the response system should be such that, depending on the current state of the network (e.g. stationary state or under DDoS attack), a list of lists and IP addresses is sent and updated live as the situation changes.

4.1 Action Scenario

The protection system described at the beginning of the chapter can be maintained for the example network shown in Figure 4.1 using the ruleset shown in Listing 4.1. At the time of the attack, an unexpectedly high volume of traffic arrives from the external network from a large number of different IP addresses, and the DDoS attack detection system reports an incident. The list of IP addresses that are considered suspicious and the ones detected as malicious is collected in a database. Depending on the number of these IP addresses and the security specifications, this may take some time. In the meantime, the DDoS attack may succeed and harm the victim.

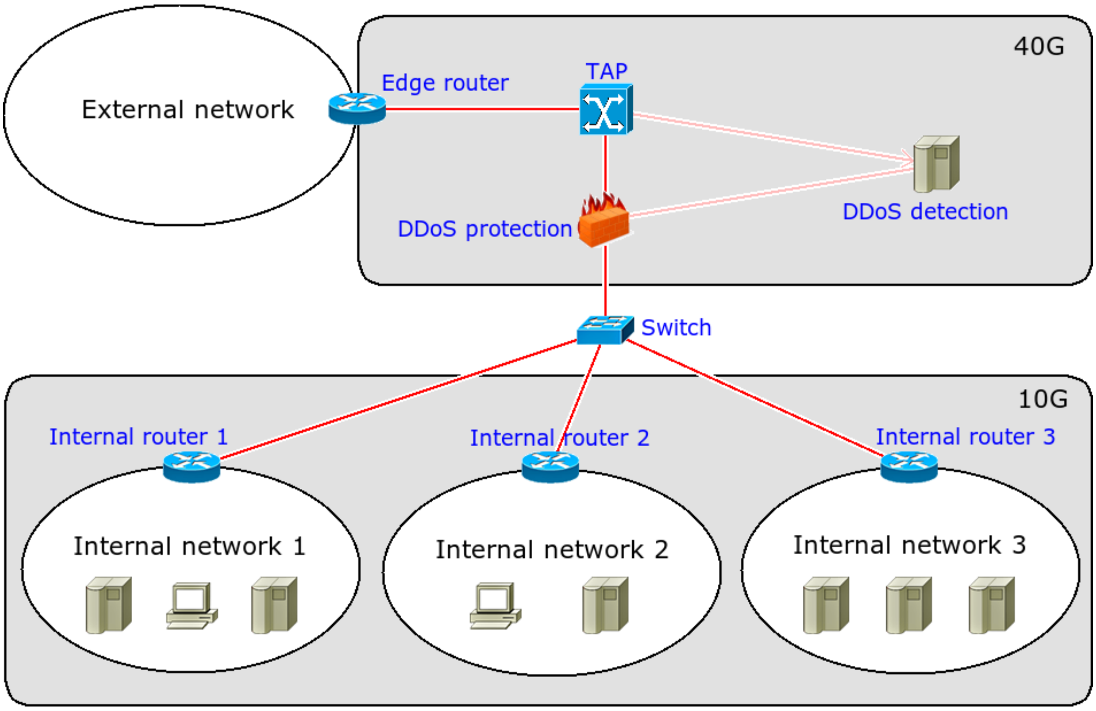


Figure 4.1: An example of a realistic network topology protected by a described DDoS mitigation system. The lighter connections represent communication within the system. The Test Access Point (TAP) device is used to copy the samples of incoming traffic to the DDoS detection node. However, the implementation of the software filter may have the option to do the same and communicate directly with a DDoS detection system without using the TAP device at all.

To avoid unnecessary downtime, immediately after the suspicious DDoS traffic is detected and depending on its threat level, some sort of allowlist can be used to forward only secure IP addresses while the external tool is collecting all suspicious IP addresses. Allowlist can be created by collecting the IP addresses earlier, during the normal operation of the network, or by using IP addresses and networks that are less likely to be dangerous (e.g., those from the address range of the country where the system is deployed) — examples shown in Listing 4.2 and Listing 4.3. In this way, the link to the victim remains open to (supposedly) secure users until a sufficient number of attacker IP addresses have been collected to reduce the impact of the attack on that link by filtering them and allowing traffic from all other addresses. When

Listing 4.1: Ruleset example that can be used with normal network traffic.

```
#Constant rules:
ALLOW hosts/nets src specific ports ADMIN # secure admin access
BLOCK hosts/nets not dst PUBLIC # non-public destinations
BLOCK hosts/nets src BAD # bad IPs/nets known from smart analysis
MONITOR hosts/nets src SUSP # suspicious IPs/nets, countries or AS
=====================================================================
Additional rules depending on DDoS level of alert:
=====================================================================
#DDoS cautious (default)
ALLOW hosts/nets src GOOD # secure IPs/nets from smart analysis
ALLOW hosts/nets src GEOIP # secure IPs/nets from GeoIP analysis
MONITOR and ALLOW everyone else
```

the system decides that the filtering removed a satisfactory amount of malicious network traffic, the filtering is switched back to a more relaxed mode, as shown in Listing 4.1. In this case, all traffic is forwarded, except for traffic defined in the blocklist (with updated "bad" IP addresses or subnets).

Listing 4.2: Ruleset example with lax allowlists.

```
#Constant rules:
ALLOW hosts/nets src specific ports ADMIN # secure admin access
BLOCK hosts/nets not dst PUBLIC # non-public destinations
BLOCK hosts/nets src BAD # bad IPs/nets known from smart analysis
MONITOR hosts/nets src SUSP # suspicious IPs/nets, countries or AS
=====================================================================
Additional rules depending on DDoS level of alert:
=====================================================================
#DDoS low alert lockdown
ALLOW hosts/nets src GOOD # secure IPs/nets from smart analysis
ALLOW hosts/nets src GEOIP # secure IPs/nets from GeoIP analysis
MONITOR and BLOCK everyone else
```

The described DDoS protection can be achieved using rulesets with only 7 rules and containing 6 LPM tables. All rulesets include the rules that would forward some secure source hosts / networks that always have access to all necessary parts of the internal network, possibly further specified by destination ports (ADMIN table — e.g. partner companies, third party administrators or employees with fixed IP addresses working from home). In addition, access to all other parts of the network that are *not* publicly accessible is blocked (PUBLIC table — e.g. IP addresses of WEB, DNS or email servers). Next, all already known bad IP addresses are blocked (BAD table — e.g. from publicly available collectors suspicious / malicious IP addresses). Finally, traffic considered suspicious is monitored (SUSP table — e.g. countries known for espionage or DDoS attacks and otherwise have no reason to access services on the internal network).

Listing 4.3: Ruleset example with only a single (more strict) allowlist.

```
#Constant rules:
ALLOW hosts/nets src specific ports ADMIN # secure admin access
BLOCK hosts/nets not dst PUBLIC # non-public destinations
BLOCK hosts/nets src BAD # bad IPs/nets known from smart analysis
MONITOR hosts/nets src SUSP # suspicious IPs/nets, countries or AS
=====================================================================
Additional rules depending on DDoS level of alert:
=====================================================================
#DDoS high alert lockdown
ALLOW hosts/nets src GOOD # secure IPs/nets from smart analysis
MONITOR and BLOCK everyone else
=====================================================================
```

The rest of the ruleset depends on the situation and may change depending on whether the network is under a DDoS attack and how severe it is. The default, shown in Listing 4.1, monitors regular traffic with an external tool and captures secure hosts / networks that accumulate in a secure table that is always forwarded (GOOD table — e.g., regular or unsuspicious users). Source addresses can be classified according to their geo-location (GEOIP table — country for which the service is intended, neighboring countries or "friendly" countries) and be included in a special table with a lower security rating. All other traffic is forwarded, but also checked by an external automatic DDoS attack detection system.

The DDoS low alert lockdown shown in Listing 4.2 is a similar state to the default state, but instead of allowing all unknown traffic, it blocks everything except potentially secure tables (GOOD and GEOIP). Since all other traffic is monitored, this helps isolate the bad IP addresses and add them to the BAD table.

If necessary, the DDoS high alert lockdown (see Listing 4.3) additionally rejects the GEOIP table if it is proven to be unsafe, but otherwise it works the same as the low alert lockdown.

4.2 Components

This hybrid approach consists of two main parts that the packet goes through (hardware — FPGA, and software — LPM filter) and an additional job distributor that determines how the packet is processed in hardware and software based on the criteria set by the user. The main task of the distributor is to decide which metadata should be sent to the hardware with the packet itself. Since the transmission of metadata in this implementation is limited by the Ethernet link between hardware and software, it is important to reduce this amount of data in order to reduce the overhead of transmission over the link. Even in the initial model that uses the PCIe bus for hardware / software communication, the amount of metadata would again be an overhead that would affect the speed of this communication, especially at higher network packet speeds.

4.2.1 Real-time reconfiguration

A DDoS attack on a target starts with full force to deprive the victim of its resources, because one of the goals of the attack is to surprise the target and not give it time to react and defend itself. For this reason, a low attack reaction time is an important property of the defense system and one of the reasons why solutions that require too much time for each ruleset change are not a good choice.

For this reason, the use of an FPGA for the hardware part of such a system seems counterintuitive, since its development cycle is complex and lengthy, and certainly not synonymous with "low response time." Development begins with the design of the system using a Hardware Description Language (HDL) such as Verilog or VHDL. Since these languages are very complex, not everyone has the skills and knowledge to update the design, even if minimal changes are required. There are a number of tools specifically developed for hardware design without the need to know these languages. They use High-Level Synthesis (HLS) and they allow the user to create hardware designs at a higher level of abstraction (e.g., using the standard C programming language) without having any knowledge of the low-level interworkings of the hardware. However, these tools are often not comparable to a human-created hardware design, especially when it comes to performance-sensitive implementations. In most cases, nuances of the hardware description language can be lost when translating from software, resulting in lower performance compared to implementations that come from regular hardware design. To create good hardware implementations with HLS, anyone using it must have a good understanding of hardware design and not treat the code like software, which defeats the main purpose of HLS.

After the hardware is described, the next steps of FPGA development must be considered: design synthesis, design verification, design implementation, and FPGA programming. These are often performed by tools that automatically "translate" the design into low-level hardware elements, verify everything (e.g., memory requirements, correct timings) and match the design with real, physical elements. This process can take up to 60 minutes or more, depending on the complexity of the hardware design and the computer on which it is performed, and is therefore not a suitable candidate for a DDoS defense system.

The prototype in this thesis is designed so that only certain memory blocks in the NetFPGA SUME need to be rewritten to change the filtering methods and metadata creation. Therefore, it is not necessary to re-implement the entire design to change the filtering configuration. Doing all the necessary memory writes take only a few seconds and is done while the system is running, not disrupting the packet filtering.

4.2.2 FPGA pipeline

Network packets enter and leave the hardware interfaces one by one. Between its ingress and egress interfaces, the packets pass through the FPGA pipeline. The pipeline consists of several modules connected in series, each of which performs a specific set of operations. The FPGA pipeline of this implementation is shown in Figure 4.2.

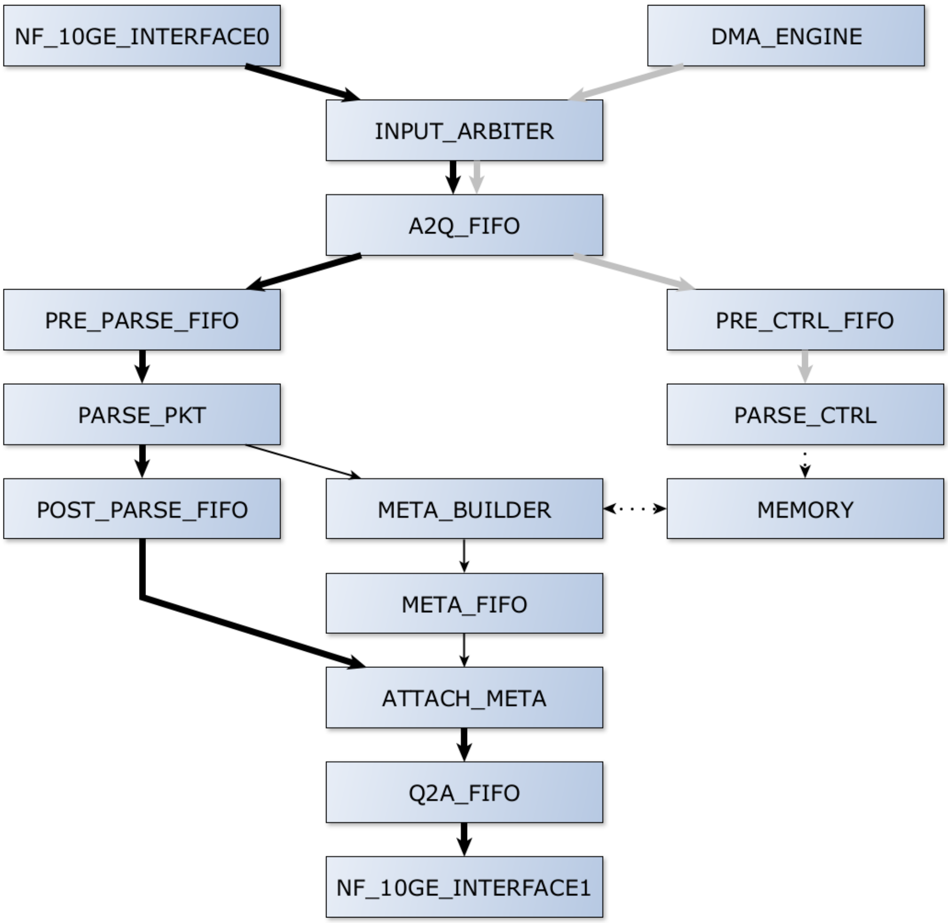


Figure 4.2: NetFPGA SUME hybrid prototype pipeline. The data transferred between modules is represented by different arrows: bold black arrows show the packet datapath using the AXI4-Stream protocol, bold gray arrows show the datapath for control packets using the AXI4-Stream protocol, dotted arrows show the paths for reading and writing memory and normal arrows show different data transfers between modules.

The network interface module (NF_10GE_INTERFACE0) converts the bits received from the optical signal transciever into 256-bit (32-byte) chunks (words) and passes them to the next modules with each clock cycle. In the FPGA infrastructure, this is implemented using the AXI4-Stream protocol with a data width of 256 bits. This protocol controls the data transfer using signals between Master and Slave modules, as shown in Figure 4.3.

On the Master side, this implementation uses the following signals:

- TDATA (output) — primary payload between the Master and the Slave, this is a 256-bit (32-byte) word of “real” network packet data.

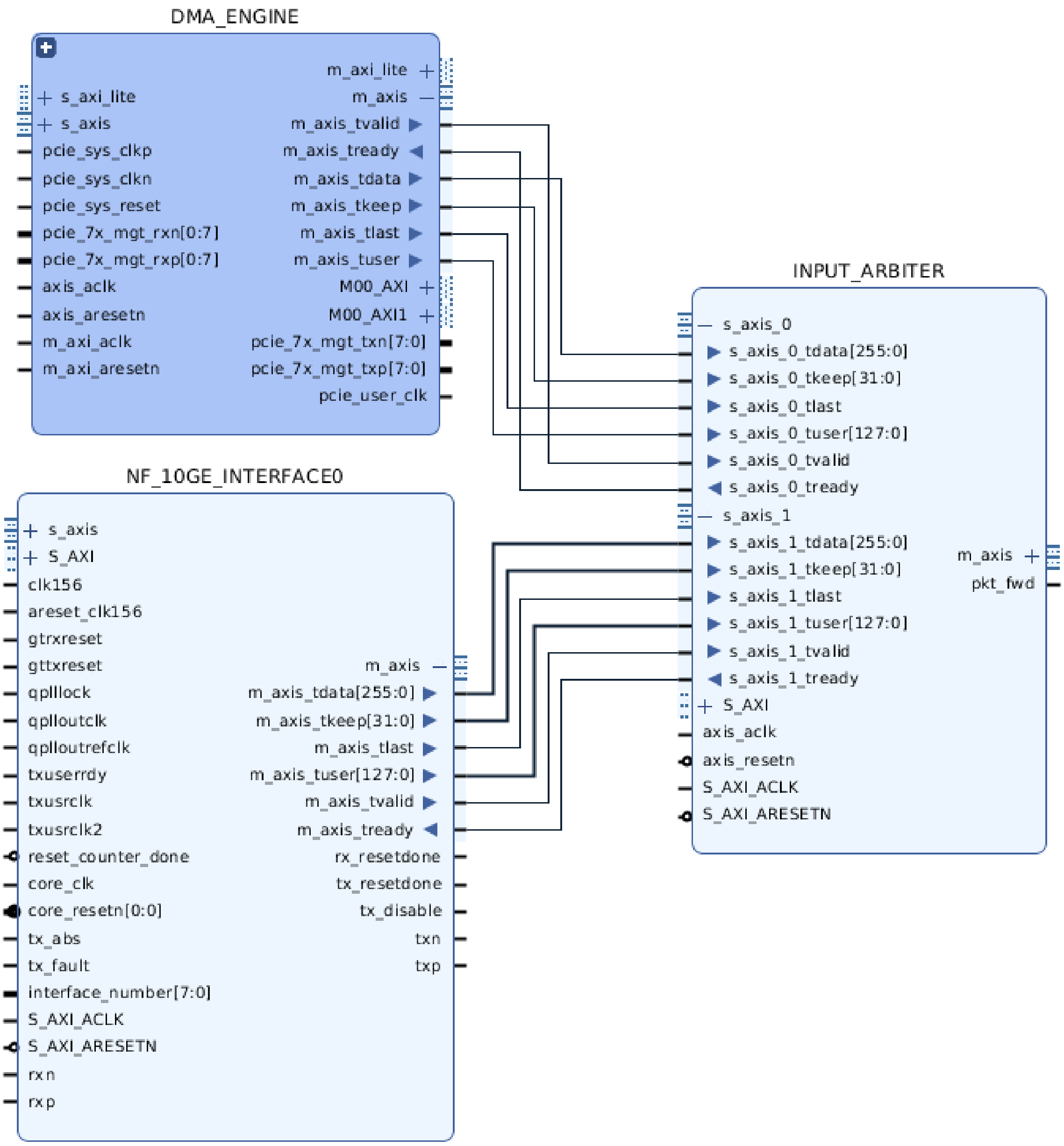


Figure 4.3: Connecting two AXI Stream Master modules (DMA_ENGINE and NF_10GE_INTERFACE0) with one Slave module (INPUT_ARBITER). The logic inside INPUT_ARBITER module determines from which Master module the data is read. All other wires are disconnected for clarity.

- TKEEP (output) — essentially an enable mask for TDATA. It is a 32-bit signal where each bit represents whether a corresponding TDATA byte contains useful data in transfer or not.
- TUSER (output) — optional user data transfered with every packet. It can be used to transfer additional data between different AXI modules inside the FPGA. This data is ignored by the physical interfaces.
- TVALID (output) — 1-bit signal which indicates that the transfer of the current cycle is valid (but only if TREADY is set at the same time).

- •TLAST (output) — 1-bit signal which indicates that the transfer of the current cycle is the last one for a packet.
- •TREADY (input) — 1-bit signal which indicates readiness of the Slave side to receive data.

Besides these signals, there are others, such as TID (stream IDentificator) or TDEST (for AXI4-Stream protocol routing information). Figure 4.4 shows an example of a two-word AXI Stream packet with each of its signals. Each word is enabled by a rising clock (CLK) edge. The 32-byte signal TDATA transmits two words (W1 and W2). The 4-byte signal TKEEP indicates two bitmaps (the first marks the entire word W0 as active, the second marks only 12 bytes of the word W1 as active). The 16-byte signal TUSER indicates two possible internal metadata words (U0 and U1). The 1-bit signal TVALID marks only two cycles as valid data. The 1-bit signal TLAST marks only the last word as the end of the packet. The 1-bit signal TREADY tells the Master module that the Slave module is ready to receive data — active only (as an example) for the first three rising edges of CLK.

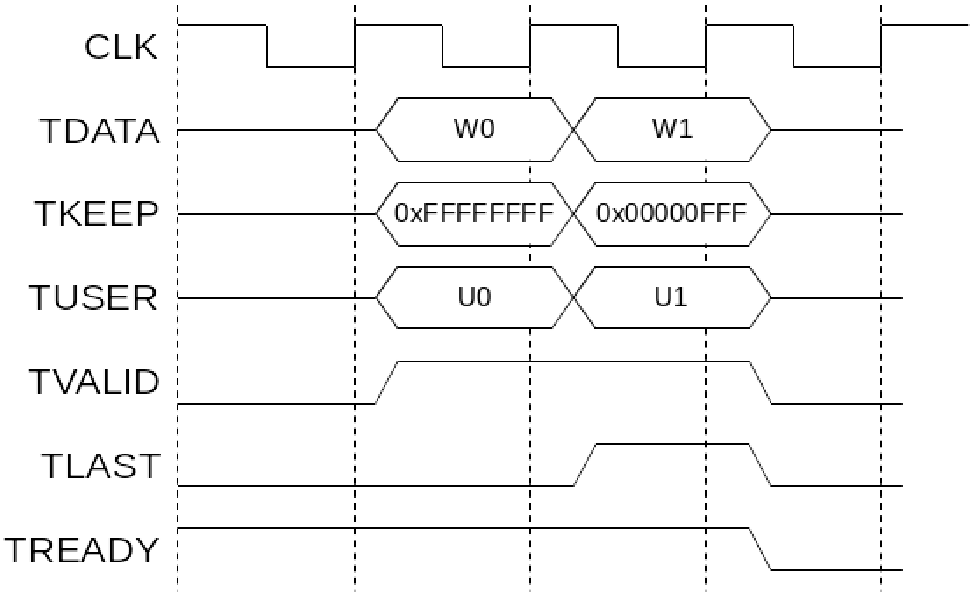


Figure 4.4: One AXI Stream packet.

In addition to the “real” network packets coming from the physical interface, packets of the same format of 32-byte data words can also come from the virtual interface on the host computer, transmitted over the PCIe bus via DMA module (DMA_ENGINE).

The next module (INPUT_ARBITER) collects the packets coming from both interfaces and forwards them further through the pipeline on a round-robin basis.

The QDR memory module used in this design requires a higher operating clock frequency than the rest of the design. If two modules need to communicate across clock domains at different frequencies, a third module is required to enable communication between them. This means that each module that communicates with the QDR module must be accompanied by a "translator" module. To simplify the hardware implementation at the current prototyping stage, the entire main part of the design is transported to the clock domain of the QDR module, using only one asynchronous FIFO module (A2Q_FIFO) as such translator. This approach could lead to potential timing issues when additional logic is added to the design. For this reason, and to reduce the overall power consumption of the NetFPGA, it is necessary to optimize the design to use an appropriate (low) clock frequency for modules that do not need to operate as fast.

After A2Q_FIFO, the pipeline is divided into two separate parts. The first part is used for packets that need to be classified and forwarded to the software. The second part is used for packets that come from the software and are used to control and manage the data flow and metadata, and to store data in the internal FPGA memory. At the output of the A2Q_FIFO module, the packet is written to one of the two standard FIFOs, depending on the type of the packet (PRE_PARSE_FIFO for real and PRE_CTRL_FIFO for control packets).

After the real network packet is read from its FIFO, the packet parsing module (PARSE_PKT) extracts all possibly useful information (e.g., Ethertype, source / destination IP address, transport layer protocol, source / destination port) and stores it in individual internal FIFOs. This information is later needed to build metadata. From the parsing module, the packet is forwarded to the next FIFO (POST_PARSE_FIFO) where it waits for the associated metadata to be built.

While the information is being parsed by the PARSE_PKT module, the metadata generator module (META_BUILDER) retrieves the required data from the available FIFOs. The META_BUILDER module creates all the metadata which will be used by the software component: e.g. Ethertype, source / destination ports or transport layer protocol. It is possible to enable different META_BUILDER fields depending on what metadata the hardware needs to include with the packet. In the current implementation, these fields are simply copied into the metadata without any processing. However, additional logic can be created to use this data to further offload filtering, as described in Chapter 3.1.

The source IP address is used slightly differently, in accordance with the LPM algorithm used in further software processing. The explanation of the part of the LPM algorithm that is important for this implementation can be found in Section 4.2.3 or in [65] in more detail. To partially offload this algorithm to hardware, the values used by the algorithm are stored in the BRAM and QDR memory used in the memory module (MEMORY). The first 16 bits of the IP address are addressed by 16-bit data from BRAM, which is combined with the next 4 bits of the IP address to obtain a 20-bit address for the next step — addressing in QDR memory. Multiple read requests are made from QDR memory (depending on how many tables are used), and the data for each table is loaded from memory. Since 72 bits can be stored to each address, 32-bit values can be stored for two different tables of the LPM algorithm, limiting the number of read requests to 2. Reading from QDR memory takes the most time (about 20 clock cycles), so when all the data is read, the metadata is created and stored in the next FIFO in the pipeline (META_FIFO).

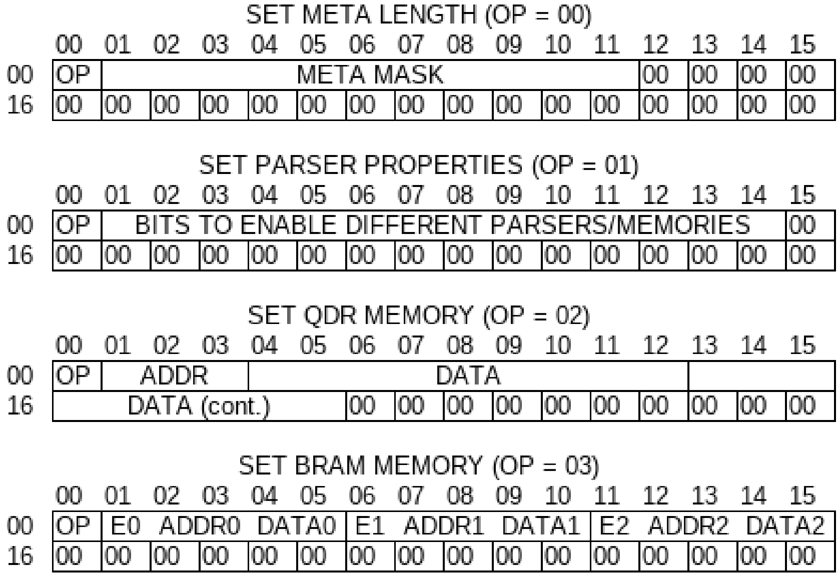
When both FIFOs (POST_PARSE_FIFO and META_FIFO) contain the necessary data, the metadata attachment module (ATTACH_META) sends the next data words until the last word of each packet is reached (when TLAST is active). If the sum of the useful bytes of the last word of the packet (from TKEEP) and the number of bytes of metadata is less than 32, the metadata is simply appended to the last word and forwarded. If the sum is greater than 32, the part of the metadata is sent with the current word and the last packet bit (TLAST) is set to 0. The remainder of the metadata is sent in the next cycle, with TVALID and TLAST set to 1, appending another word to the packet.

Before the data stream reaches the output interface, it is converted back to the low frequency clock domain by the Q2A_FIFO module. The module forwards the data to the output network interface module (NF_10GE_INTERFACE1) and the transceiver, from where it is forwarded to the software filter NIC via the optical link.

The pipeline path for control packets (the right side in Figure 4.2) includes PRE_CTRL_FIFO, PARSE_CTRL and MEMORY modules. The data from PRE_CTRL_FIFO is read by the module for parsing control packets (PARSE_CTRL). It determines the command contained in the packet and executes it. The control packets are used for:

- setting the metadata length,
- enabling / disabling individual parsing modules,
- writing to QDR memory (one QDR module with 2 ^{20} 72-bit memory locations, giving the total of 9 MiB of memory) or
- writing to BRAM memory (two separate BRAM tables with 2 ^{16} 16-bit memory locations, giving the total of 256 KiB of memory).

As mentioned earlier, each packet (including control packets) is sent through the pipeline in 32-byte words in one clock cycle. In this implementation, a command and its parameters can fit into a 32-byte word, allowing multiple commands to be sent in one packet. Sending a smaller number of large control packets speeds up data transfer from the host computer to the FPGA. When a large number of control packets needs to be sent, such as when filling all memory locations, this speedup is quite noticable, as the packets are transmitted over the PCIe and the bottlenecked NetFPGA SUME DMA engine. The maximum packet size (the MTU of the NetFPGA SUME virtual interface is 1500 bytes) can hold 46 32-byte words of data, which means that $2^{20}/46 \approx 23,000$ packets are needed to fill all 2^{20} QDR memory locations, and $2^{16}/46 \approx 1,400$ packets are needed to fill all 2×2^{16} BRAM memory locations (the data for both BRAM tables are stored in the same cycle). It is possible to send the data from the host computer and fill all the memory locations in a very short time, even if a slow SUME driver with a low transfer rate is used. This is done without affecting the performance of the filter. The formats of the control packets are shown in Figure 4.5.



SET META LENGTH (OP = 00)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	OP	META MASK											00	00	00	00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

SET PARSER PROPERTIES (OP = 01)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	OP	BITS TO ENABLE DIFFERENT PARSERS/MEMORIES														00
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

SET QDR MEMORY (OP = 02)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	OP	ADDR			DATA											
16	DATA (cont.)						00	00	00	00	00	00	00	00	00	00

SET BRAM MEMORY (OP = 03)

	00	01	02	03	04	05	06	07	08	09	10	11	12	13	14	15
00	OP	E0 ADDR0 DATA0					E1 ADDR1 DATA1					E2 ADDR2 DATA2				
16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00

Figure 4.5: Formats of the control packets. It is possible to have multiple BRAM tables — in this image, the format for three of them is shown on the last example (DATA0, DATA1 and DATA2).

Setting the metadata length (OP = 00) copies the mask from the incoming packet to the global register available to other modules that use it. Setting the parser properties (OP = 01) sets the bits that enable / disable individual models for packet parsing. Writing to QDR memory (OP = 02) is a bit more complicated, as a connection must be made to the QDR memory module, and the address to which the data is to be written must be sent in addition to the data itself. Other parts of the logic for writing to QDR memory are built into the implementation itself and there is no need to send anything else in the control packet (it is possible to add another QDR memory module to the implementation and allow parallel read / write to two QDR memory modules). To store data on BRAM memory (OP = 03), it is also necessary to specify the address where the data should be stored. It is possible to use multiple BRAM tables to store data in parallel, during the same clock cycle. The E0, E1, etc. fields must be set to a non-zero value for the data to be actually written, depending on which of the tables the user wants to write data to.

The utilization of NetFPGA SUME resources is shown in Figure 4.6. The key elements for the design of this prototype are LUT (LookUp Tables), LUTRAM (distributed Random Access Memory made out of LUTs) and BRAM (dedicated Block RAM), memory types on the FPGA that are used in the implementation of the design. LUT and LUTRAM are mainly used for logic. Since only about 8% is used, it is obvious that there is enough space to extend the existing design. As described earlier, BRAM is used for data storage. With 38%, there is also room to extend the implementation with other elements, such as IP address matching for a limited number of rules.

The FPGA design of the implementation described in this thesis can be used with any com-

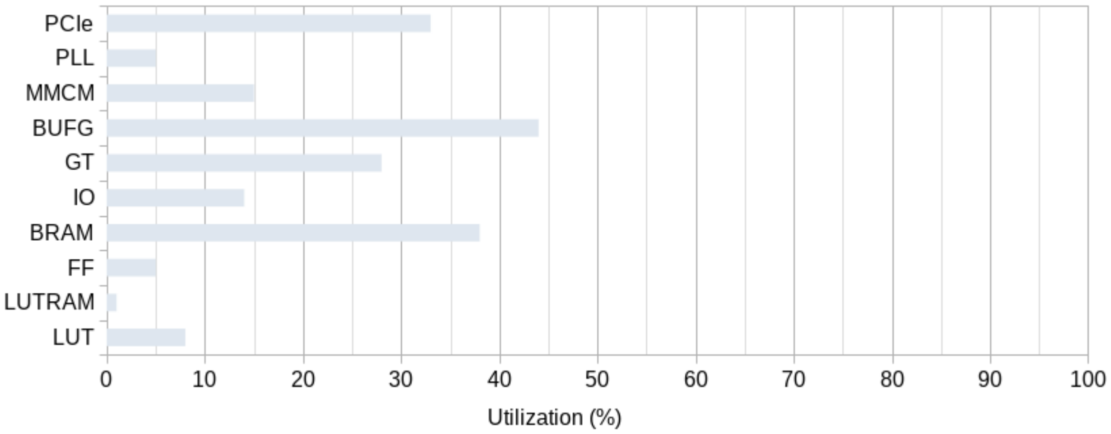


Figure 4.6: Resource utilization of the implemented design on the NetFPGA SUME, data exported from Vivado 2020.2 tool.

patible FPGA board equipped with the components required to operate as a network device. Apart from the external memory module (QDR module in this implementation), the design itself is generic and does not use any special hardware components.

4.2.3 Software filter

The basis for the software part of the system implementation is the Reduced Feature-Set Packet Filter (RFPF), described in [70], used for filtering IPv4 traffic. Although it was primarily developed for FreeBSD OS, it was later ported and adapted for Linux OS. After tuning and tweaking, the Linux version performed slightly better, perhaps due to the author's familiarity with the Linux OS. The filter uses the *netmap* framework to insert a user-defined datapath between two physical network interfaces on a general-purpose computer.

Each packet from both interfaces is checked and forwarded or dropped according to the ruleset specified. The RFPF tool is a powerful and reliable high-speed stateless firewall written in the C programming language. It parses a ruleset written in plain text format and generates new C code from it, which is compiled into a dynamically linked object and "inserted" into the datapath at runtime. The use of *netmap* and additional compiler code optimizations enabless high-speed packet processing that can forward traffic at 10G speeds with as little as one CPU core.

Parsing, generating, and inserting the ruleset into the "live" datapath can be relatively fast (this depends on the size of the ruleset and the number of prefixes in the tables used, as well as the CPU frequency and load) — the largest ruleset used in this thesis was ready in less than 20 seconds. Even this meets the criterion for a low response time in case of a DDoS attack, but it is also possible to hot-swap the enabled datapath ruleset with a pre-loaded one for even faster response times (2–3 seconds). Until the new ruleset is fully loaded, the filter will continue to work with the “old” (current) ruleset and will not drop any packets.

When filtering with allowlist / blocklists, the LPM algorithm is used to extract data from each list to determine whether or not the IP address being checked is in a particular list. Any LPM agorithm can be used for this task, but the implementation of the software filter used in this thesis uses the DXR algorithm from [65, 69], more specifically the D16X4R version. The DXR algorithm works by compressing and storing a list in compact structures with a small memory footprint. The original algorithm consists of 3 stages (D16, X4, and R), all executed in software. The modified version used in this implementation separates the first two stages (D16 and X4 — indexing and retrieving from memory using a total of 20 bits) and executes them in hardware. The hardware then forwards the result of these two stages (the 32-bit index and a range) to the software. In the last stage, a binary search is performed over the received range (R — range lookup) until the algorithm reaches its end and returns the final result. The final result is the “next-hop” for a given LPM table (i.e. a label from this table), or a null value indicating a non-match.

The LPM algorithm used in this thesis is not the only one that can achieve high performance LPM lookups, but it was chosen to create a proof-of-concept prototype due to the author’s familiarity with it and it showed promising results for IP routing when researched for this thesis. Other modern algorithms include PopTrie [66], SAIL [67], but there are also older algorithms, such as DIR–24–8 [90], that can be used in a similar way. DXR has been shown to be superior to other algorithms in software filtering, but some of the algorithms listed could work just as well (or even better) in the proposed hybrid model.

The original idea was to prepend the metadata at the beginning of the packet so that it would always be in the same place. However, to simplify the implementation, it was decided to append it to the end of the packet. When using the *netmap* framework, the program is given the pointer to the beginning of the packet and its total length, so with the known length of the metadata, the pointer to the position of the metadata can be easily calculated.

Special attention was required in adapting the existing software filter to the hybrid mode of operation. The new software implementation had to consider which parts of the packet were included in the metadata and how the metadata could be automatically integrated into the generated C code after parsing the ruleset. An example of a simple generated code-snippet can be seen in Listing 4.4, where it is possible to save CPU cycles on one `ntohl` operation and even before that when calculating the position of the IP header.

4.2.4 Distributor

As already mentioned, before the distributor can divide the work, some assumptions must be taken into account:

Listing 4.4: Part of the code generated by RFPF, showing the difference between offloaded and non-offloaded code.

```
#if OFFLOAD & OFF_IP
    // if IP addresses are parsed from the metadata
    uint32_t ip_src = meta->src;
#else
    // if IP addresses are not parsed from the metadata
    uint32_t ip_src = ntohl(ip->ip_src.s_addr);
#endif
```

- traffic type — what type of traffic is expected to go through the filter and what type of traffic should be dropped,
- traffic volume — for lower volume traffic, the filter can do most of the work in software without utilizing the hardware, saving power when the FPGA is offline,
- security conditions — different rulesets should be "activated" depending on the state of the network: is it under DDoS attack or not,
- rule categories — some rules can be implemented in hardware and in software, while other rules can only be implemented in software,
- rule number — hardware memory may not be large enough for all rules, so some rules should be done in hardware, but others in software,
- network topology — some parts of the network can be more important, so those rules should have priority.

Other factors that may also affect offloading are hardware / software limitations and user-given specifications:

- software: filtering implementation, CPU frequency, CPU cache, cache size, etc.,
- hardware: filtering implementation, FPGA clock speed, memory size, memory type, etc.,
- certain rules can be forced to be (non-)offloaded, etc.

In addition to these constraints and assumptions, there is also the possibility of "minimizing" or "simplifying" the ruleset before it is offloaded (e.g., by implementing an appropriate PCE from Section 2.2), so that the filter can reduce resource consumption at both hardware and software.

Since any change to the hybrid model may also require changes to the distributor model, the distributor in the current version of prototype is not implemented as a tool, but is defined as a heuristic method for selecting the best ruleset / metadata combination depending on pre-specified assumptions, parameters and factors.

For example, in this thesis, it is assumed that the ruleset is created to protect the network from volumetric DDoS attacks with a large number of random source IP addresses. To get the most out of the LPM algorithm, the large lists of IP prefixes (such as allowlists and blocklists) are used to generate LPM tables that are maintained separately from the rulesets. As a result, the number of rules in the ruleset is much smaller than in standard firewalls. These rules are mostly category P_1** or $N**$. Under this assumption, the hardware part of the implementation is mainly created to leverage the structures of the LPM algorithms, although it is possible to repurpose it for other types of metadata. Section 4.2.2 explains in detail how the hardware implementation works. The distributor model for this implementation is described in Figure 4.7 and is derived from the performance evaluation in Chapter 5.

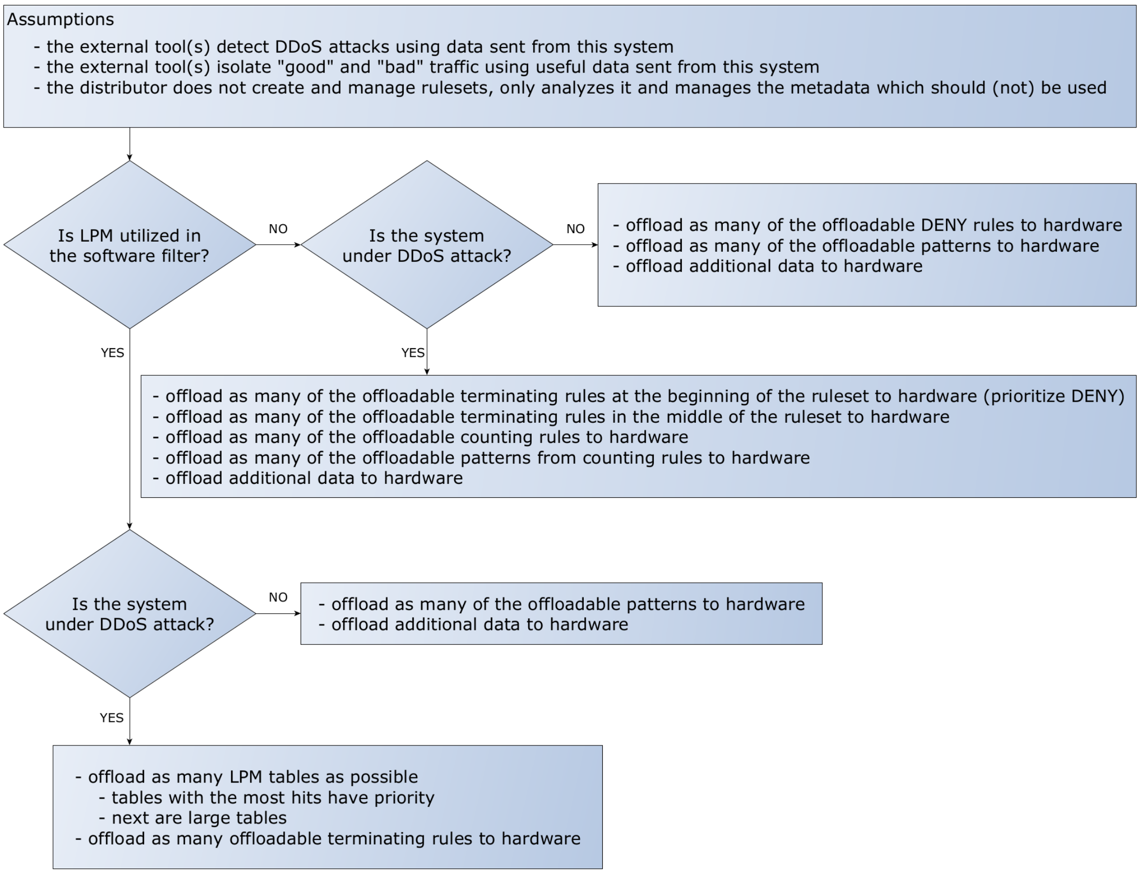

Assumptions
- the external tool(s) detect DDoS attacks using data sent from this system
- the external tool(s) isolate "good" and "bad" traffic using useful data sent from this system
- the distributor does not create and manage rulesets, only analyzes it and manages the metadata which should (not) be used
Is LPM utilized in the software filter?
NO
Is the system under DDoS attack?
NO
- offload as many of the offloadable DENY rules to hardware
- offload as many of the offloadable patterns to hardware
- offload additional data to hardware
YES
YES
- offload as many of the offloadable terminating rules at the beginning of the ruleset to hardware (prioritize DENY)
- offload as many of the offloadable terminating rules in the middle of the ruleset to hardware
- offload as many of the offloadable counting rules to hardware
- offload as many of the offloadable patterns from counting rules to hardware
- offload additional data to hardware
Is the system under DDoS attack?
NO
- offload as many of the offloadable patterns to hardware
- offload additional data to hardware
YES
- offload as many LPM tables as possible
- tables with the most hits have priority
- next are large tables
- offload as many offloadable terminating rules to hardware

Figure 4.7: The distributor model for the hybrid filtering system.

4.3 Simulations

Packets received by the software filter component already have metadata attached to them (generated by the hardware component). To test the software component, it was easier to send the packets directly to it without using the hardware, as if the packets came from the hardware with metadata already added. To do this, the packets and metadata were generated using the software packet generator without having to constantly change the FPGA design, code in complex HDL, and wait for bitstream synthesis. Simulating the hardware in this way bypassed the complex and time-consuming design and implementation and eliminated the middleman during testing.

This avoided potential bugs and tedious troubleshooting when the packets sent by the generator would not match those that arrived at the destination. At the same time, the evaluation of the offloaded filtering proved to be more flexible and easier to perform.

The simulated packets are generated using the *pkt-gen* tool included in the *netmap* framework, which is otherwise used to generate network packets from the network interface at high transmission speeds. It had to be modified to create the metadata fast enough when generating random packets and to include it completely in the packet when needed. Since the *pkt-gen* was working on a different computer, it was possible to use multiple cores to achieve sufficiently high transmission speeds without affecting the throughput of the filter.

Chapter 5

Benchmarks and performance evaluation

Due to the limitations of this implementation mentioned above (additional transmission of metadata along with the packets themselves, i.e., using part of the bandwidth between hardware and software exclusively for metadata), it is not possible to fully utilize the bandwidth of incoming traffic at 10G speeds. Therefore, comparisons between offloaded and non-offloaded filtering are performed at lower speeds to avoid reaching the upper limit of the network interface. The maximum packet processing speed at which tests can be made depends on the size of the metadata. For example, with minimum packet sizes and a metadata size of 18 bytes, a processing speed of about 12.3 Mpps can be achieved. All tests are set so that the bandwidth never reaches the maximum value. To allow better control and consistency of tests, only one CPU core with reduced frequency is used[1].

In this way, the efficiency of the two filtering methods can be compared based on the number of packets processed per second and the number of CPU cycles required to process one packet. In order to avoid possible deviations in the average results due to idle CPU and to take full advantage of it, CPU was always 100% busy during the tests. It is important to note that NetFPGA SUME can forward minimum size packets (with metadata size of 0 bytes) from one interface to another at a 10G line rate (14.88 Mpps) using the pipeline from the described implementation. This shows that the operation of pipelined elements does not affect the overall throughput, but is only limited by the size of the metadata due to constraints of NetFPGA SUME in this implementation (as explained in Section 3). To take full advantage of it, NetFPGA SUME would need to be used as a NIC, along with a DMA engine and a driver capable of higher throughput.

To compare the impact of different offload types on processing and filtering, tests use throughput (given in Mpps) and the average number of cycles the CPU takes to process one

[1] Software filter computer has an Intel Core i7 10700K CPU (@3.7 GHz), 16GB of DDR4@3200 MHz RAM, Asus Z490-P motherboard and uses a dual-port Intel X520 10G NIC. Tests are performed at a CPU frequency of 2.2 GHz. The generator / sink machine has an AMD Ryzen5 3600XT CPU (@3.8 GHz), 16GB of DDR4 3200MHz RAM, Asus Prime X570-PRO motherboard and uses the same type of Intel X520 10G NIC.

packet.

Each rule of the software filter used in this thesis has a built-in counter for all matched packets. The filter uses a global counter for all incoming traffic and measures a 3-second and a 60-second averages for total throughput. The 3-second average is used to verify that the throughput has a stable value with no fluctuations during the tests. The test results are based on the more accurate 60-second average. This throughput value is verified in the traffic sink, which also calculates the throughput of incoming packets in 1-second intervals. The CPU cycle counter is also implemented in the software filter, using the assembler instruction *rdtsc* [91], which acquires the processor's timestamp counter before and after processing each batch of packets. Using these values, it is possible to calculate the average number of cycles required for one packet in each 3-second and 60-second interval.

5.1 Methodology

As explained in Section 4.3, the simulation of the hardware was done with metadata inserted by an external software tool. This was the first step in testing the results for offloading different types of metadata, rulesets and traffic types without actual hardware. This allowed much greater flexibility in experimenting with different metadata types, sizes, and structures.

A software filter without hardware offloading is used as the baseline test for each ruleset. The packet generator sends "normal" traffic consisting exclusively of packets without metadata. The software filter receives the traffic, performs the necessary processing, and forwards or drops the traffic without modifying the packet. For the same ruleset, multiple tests were performed for offloaded filtering: changing the metadata and the type of traffic, and comparing the results with the corresponding baseline result.

All simulations were run on the same testbed (shown in Figure 5.1), and the sessions for each ruleset were measured separately.

It has been observed that repeating one type of test for the same session does not give consistent results in terms of performance. Applications that use the *netmap* framework (software filter and *pkt-gen*) "hook" to interfaces at startup and "unhook" to them at shutdown. This makes *netmap* prone to minor inconsistencies that show up when comparing different results from the same test. The results for the same type of offload were noticeably different each time the test was repeated, so these inconsistencies had to be addressed. To avoid them in individual tests, the *netmap* applications always remained online during each test. Although this proved effective in avoiding result fluctuations, further research is needed to understand and fix the issue.

To keep *netmap* applications online, two major changes were implemented. The first was to modify the software filter to allow multiple different offload configurations to be stored without

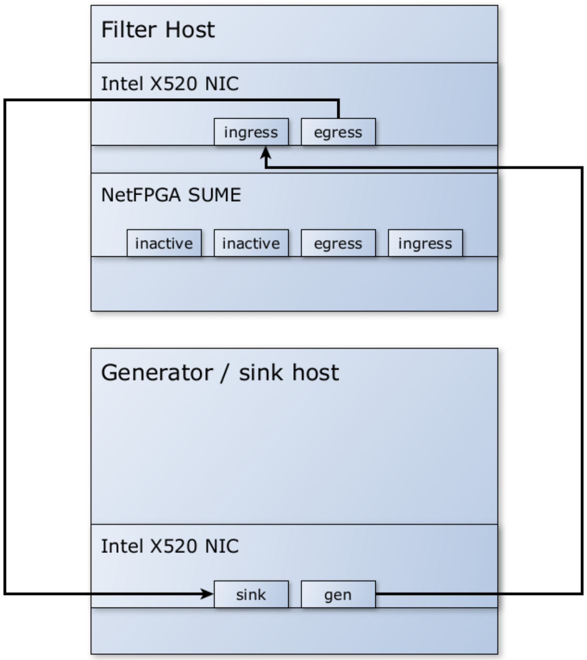


Figure 5.1: The testbed for simulated hardware tests, bypassing the NetFPGA SUME.

having to change and rebuild the generated C code. The second was to modify *pkt-gen* to dynamically change both the metadata sent and the type of traffic generated without having to restart it each time. These changes allowed each test to be consistent throughout the session, but each time the session was restarted, the results were slightly different again. For this reason, multiple tests were run for each session to calculate average values for throughput and number of cycles.

Because of to the way the software filter and packet generator were implemented, the tests were initially performed by:

- turning on the sink (it can be left running for all sessions),
- measuring the results for non-offloaded filtering:
  - loading the current session ruleset with the software filter,
  - turning on the packet generator,
  - waiting for the throughput to stabilize,
  - recording the throughput and cycle count,
  - stopping the generator,
- measuring the results for offloaded filtering — repeated multiple times for several different types of offloads, depending on the session:
  - changing the offload type for the software filter,
  - rebuilding the dynamically loaded filter configuration,
  - turning on the packet generator with the changed metadata,
  - waiting for the throughput to stabilize,

  - recording the throughput and cycle count,
  - stopping the generator,
- stopping the filter.

The same tests were performed twice: for `random` traffic and for `specific` traffic.

After the modifications of the *netmap* applications, the new testing method was slightly different:

- turning on the sink (it can be left running for all sessions),
- loading the current session ruleset with the software filter (no offload),
- saving the configuration,
- changing the offload type for the software filter — repeated multiple times for different types of offloads, depending on the session:
  - setting the offload type for the software filter,
  - loading the current session ruleset with the software filter,
  - rebuilding the dynamically loaded filter configuration,
  - saving the configuration,
- turning on the packet generator,
- measuring the results for every type of offload (including no offload):
  - loading the next saved filter configuration,
  - changing the metadata for the traffic generator,
  - waiting for the throughput to stabilize,
  - recording the throughput and cycle count,
  - switching the generator traffic,
  - waiting for the throughput to stabilize,
  - recording the throughput and cycle count,
- stopping the filter,
- stopping the generator.

All the parameters (metadata types, rulesets and traffic types) used in tests are explained below.

5.1.1 Metadata types

Packet filtering is tested using the metadata types described in Section 3.1.5, combining multiple metadata for some tests:

- `metadata1` — data used in partially offloaded processing. Combinations of protocol, IP / port source / destination, and 1/2/4 LPM tables offload were tested,
- `metadata2` — a bit for every p_O *pattern* from partially offloaded rules $P_0 * *$ or $P_2 * *$. Metadata sizes of 8/16/32/64/128 bits were tested,

- `metadata3` — a bit for every rule with the *Count* attribute. Metadata sizes of 8/16/32/64/128 bits were tested,
- `metadata4` — matched rule information. Metadata size for these tests always stayed the same (8 bits). Different number of *terminating* rules (8/16/32/64/128) that can be offloaded to hardware was tested.

5.1.2 Ruleset types

The rulesets used in tests are divided into two groups, depending on the type of rules used in them. One group uses only *COUNT* rules, forcing the software filter to process each of them before forwarding the packet to the egress interface. This ensures that the same number of processing operations are performed for each packet, making the speed of filtering comparable for all tests for the same group. The second group uses *terminating ACCEPT* rules in addition to *COUNT* rules. If the packet matches the *terminating* rule, it no longer needs to be processed, so subsequent rules are not checked. For this reason, the throughput for these tests is slightly higher than the throughput for tests from the first group of rulesets. The results of tests from the first group should not be compared with the results of tests from the second group.

It has been observed that for rulesets with a small number of rules (i.e., with fewer prefixes in total), even the baseline filtering can reach the maximum throughput without the CPU load reaching 100%. For this reason, no comparison was performed for such rulesets.

All of the following rulesets belong to the first group, i.e., every rule in the ruleset must be processed:

- `rs-ip` — 1000 rules of the same format. Each rule checks a random source / destination IP address — uses `metadata1`.
- `rs-headers` — 1000 rules of the same format. Each rule checks a random source / destination IP address, a protocol and a source / destination port — uses `metadata1`,
- `rs-partoff` — 1000 rules of the same format. Each rule checks a random source / destination IP address, a protocol and a source / destination port range — uses `metadata2`,
- `rs-fulloff` — 1000 rules of the same format. Each rule checks a port range — uses `metadata3`,
- `rs-lpm` — 9 rules of the same format. Each rule checks whether a source IP is stored in one of the 9 LPM tables — uses `metadata1`,
- `rs-mix` — 1009 rules of mixed formats. Rules from `rs-lpm` randomly inserted rules into `rs-headers` ruleset — uses `metadata1`,
- `rs-lpmpart` — 9 rules of mixed formats. All rules are the same as the ones from the `rs-lpm` ruleset, but 8 of them additionally check if the destination port is within a certain port range — uses `metadata1` and `metadata2`,
- `rs-lpmfull` — 137 rules of mixed formats. Rules from `rs-lpm` randomly inserted into

128 of the fully offloadable *COUNT* rules that check if the destination port is within a certain port range. — uses `metadata1` and `metadata3`.

All the following rulesets belong to the second group, i.e., they contain *terminating* rules:

- `rs-skipa` — 1000 rules, each rule checks a random destination port range, 128 of the rules are *ACCEPT* — uses `metadata4`
- `rs-skipa_alt` — 1000 rules, same as `rs-skipa`, but the *ACCEPT* rules are at the beginning of the ruleset — uses `metadata4`
- `rs-lpmskipa` — 137 rules, same as `rs-lpmfull`, but non-LPM rules are *ACCEPT* instead of *COUNT* — uses `metadata1`, `metadata4`
- `rs-lpmskipa_alt` — 137 rules, same as `rs-lpmskipa`, but the *ACCEPT* rules are at the beginning of the ruleset — uses `metadata1`, `metadata4`

5.1.3 Traffic types

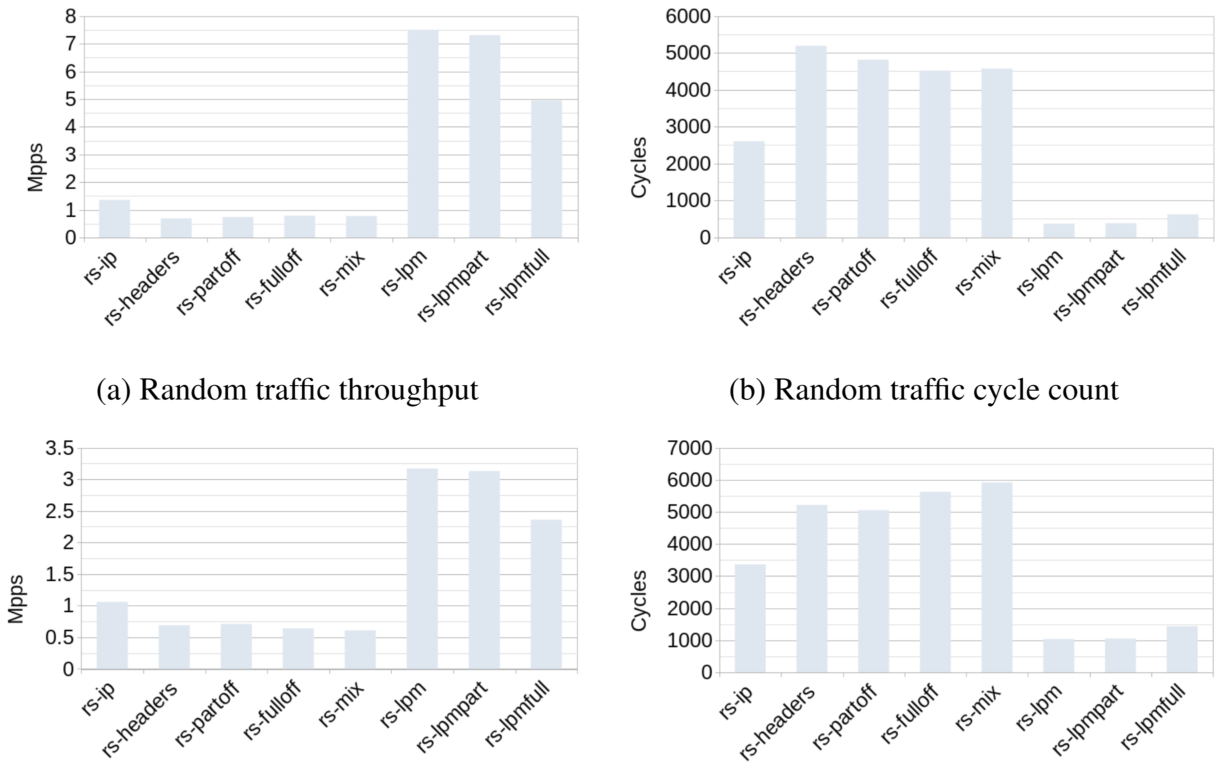
Since it is very difficult to obtain real DDoS traces because ISPs and larger companies that have been victims of attacks are reluctant to release them (mainly for privacy reasons), this thesis uses synthetic traffic as the source of DDoS traffic. Different types of inbound traffic affect the performance of the filter differently. For example, traffic with repetitive characteristics puts much less load on CPU than `random` traffic because in those cases CPU is more likely to utilize its cache and instruction prediction. Matching multiple different rules also requires more CPU cycles.

Two different types of traffic are used for tests:

- `random` — completely random traffic, random source / destination IP addresses and ports,
- `specific` — DDoS traffic with `specific` source / destination IP addresses and ports (in order to match most of the rules in the ruleset) mixed with 10% random traffic.

Completely random traffic is not very realistic, either in a DDoS attack or as everyday traffic, but it serves as a benchmark for CPU testing with unexpected inputs and non-cacheable results. Random traffic mixed with mostly malicious traffic with IP addresses from blocklists and with regular traffic from allowlists is a much more realistic case of a DDoS attack. This shows how well the filter can work in situations where most of the traffic matches the rules of the ruleset. The `random` traffic was generated from the entire set of IP addresses (2^{32} source IP addresses in total). The DDoS (`specific`) traffic covers a narrower range of IP addresses, depending on the ruleset of the particular test.

Figure 5.2 shows the average throughput and average number of cycles for all rulesets in the first group for both types of traffic. Figure 5.3 shows the average throughput and average number of cycles for all rulesets in the second group for both types of traffic.



(a) Random traffic throughput

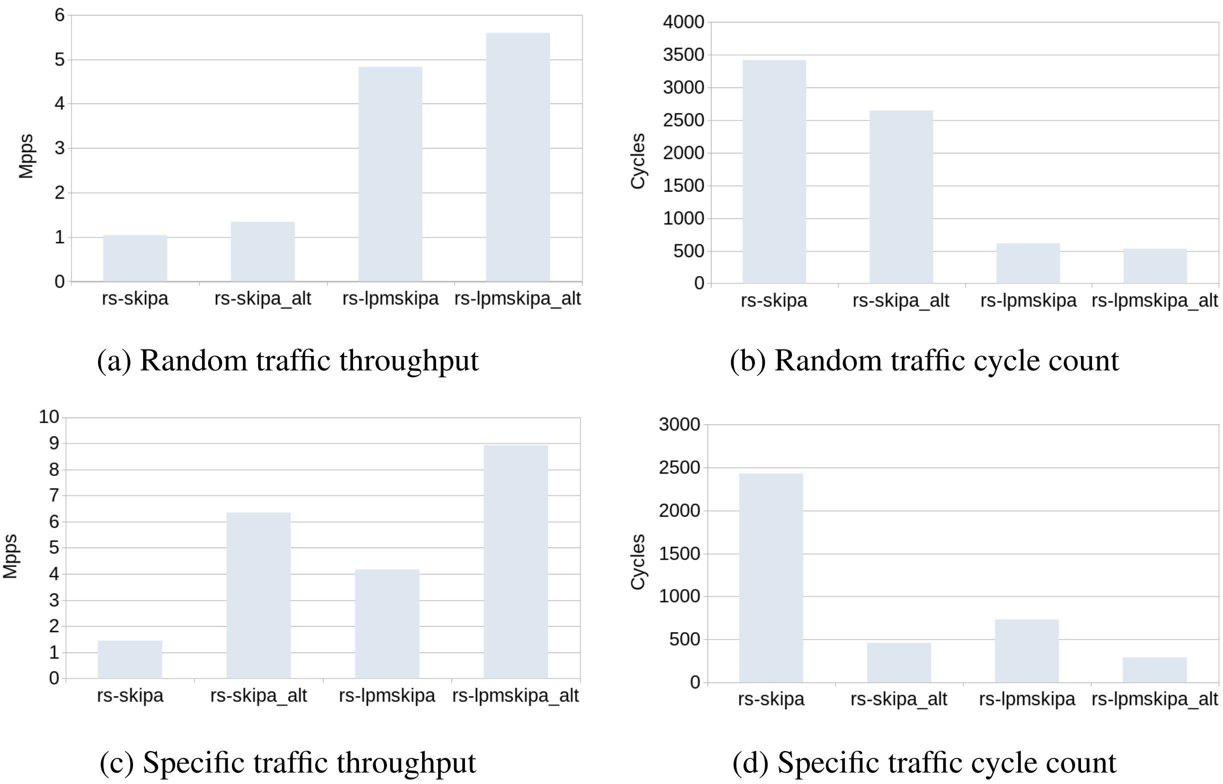
(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.2: Average filtering throughput and average cycle count of non-offloaded rulesets (rulesets with only *COUNT* rules).

In Figures 5.2 and 5.3, there is a significant difference in the throughput of one type of rulesets (without LPM) compared to the other type (with LPM). This difference highlights the advantage of using LPM when filtering traffic in high-speed networks. The results of these two types of rulesets are not comparable because they differ in the total number of prefixes used when filtering. To make the comparisons with the software filter implementation from this thesis truly “fair,” the non-LPM rulesets would have to be extremely complex and large (degrading throughput even more), or the LPM rulesets would have to be extremely simple (with even higher throughput). A similar method to LPM, but inferior in performance (due to the large memory footprint), is used by the *IPset* utility, which stores and lookups prefixes to a set using hash functions.



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.3: Average filtering throughput and average cycle count of non-offloaded rulesets (rulesets with COUNT and ACCEPT rules).

5.2 Software filter with pre-generated metadata

The rulesets and their test results are shown in this section. All results from the evaluation show the comparisons of different offloaded filtering results (throughput and CPU cycle count) with non-offloaded (baseline) results.

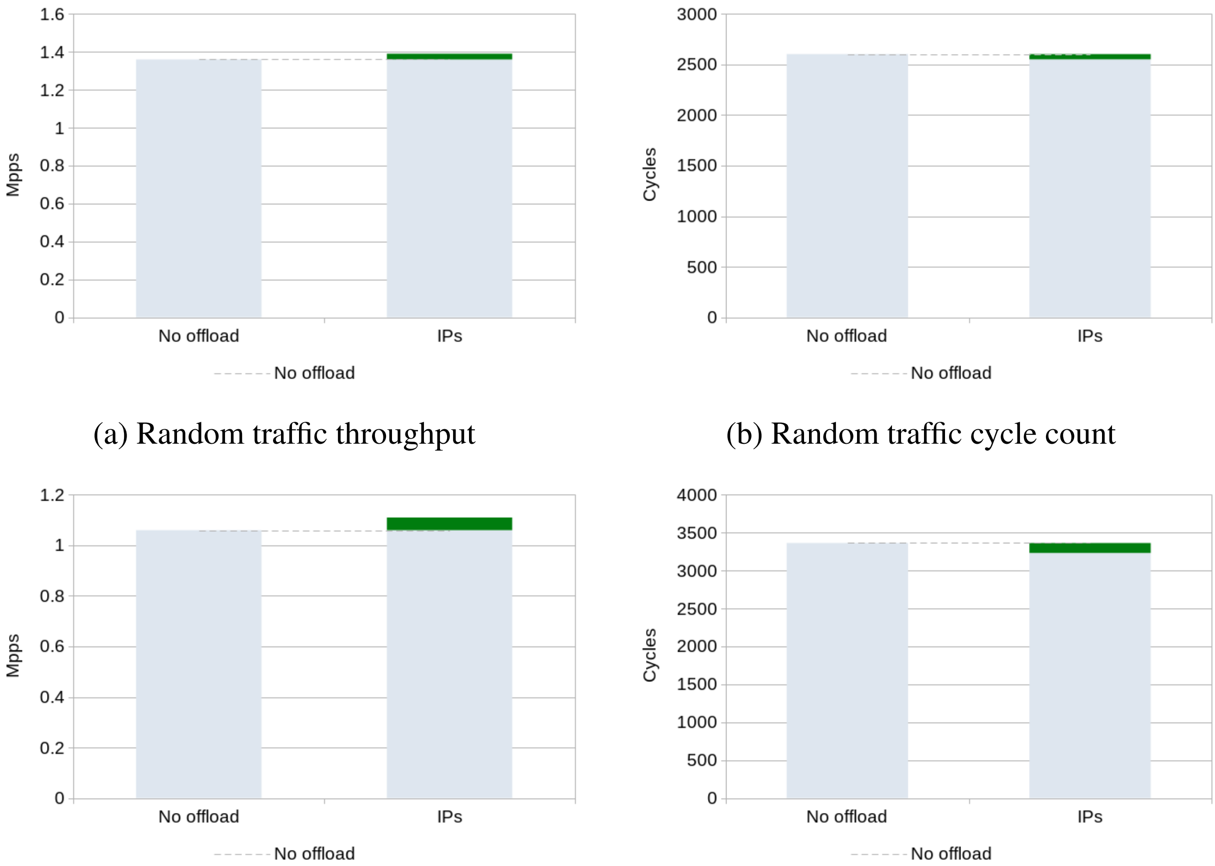
Ruleset `rs-ip`

To combat volumetric DDoS attacks, a large number of IP addresses must be filtered. Listing 5.1 shows part of a simple firewall, filtering only 1000 different IP addresses. The ruleset consists only of simple rules to check the source / destination IP address of the packet. The tests are performed under the assumption that each *pattern* is *Partially offloadable* (p_P), which puts the rules in the P_1 category. The IP addresses were inserted as `metadata1` in the metadata to be used by the software filter, as explained in Section 3.1.5.

The results in Figure 5.4 show an average increase in throughput and a decrease in the number of CPU cycles of about 3.9% for `specific` traffic. The improvement for `random` traffic is slightly smaller at about 2%.

Listing 5.1: Excerpt from the ruleset `rs-ip`. All IP addresses are randomly generated.

```
...
count src 110.10.179.196
count src 141.134.184.89
count src 182.34.201.184
count dst 156.54.137.236
count dst 207.244.118.207
count src 106.75.133.10
count dst 191.6.91.22
count dst 83.209.148.98
...
```



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.4: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-ip`.

Ruleset `rs-headers`

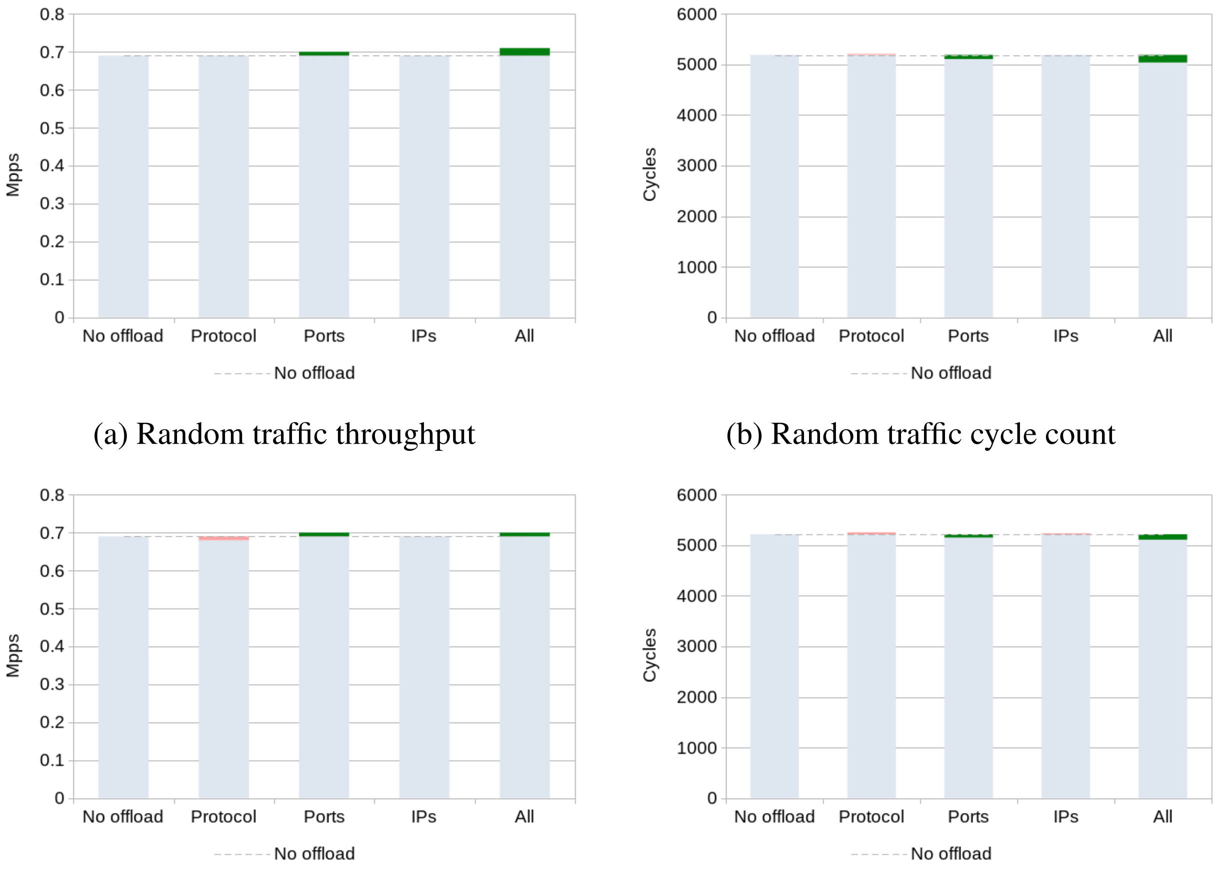
To further test the influence of the different `metadata1` information on filtering, a ruleset with multiple test parameters was created. In addition to the source / destination IP addresses from the `rs-ip` ruleset, each rule also contains the source / destination ports and protocol. Each rule is still categorized as P_1, and each check can be defined as a separate *Partially offloadable pattern* (p_P). Part of the ruleset can be seen in Listing 5.2.

Listing 5.2: Excerpt from the ruleset `rs-headers`. All IP addresses, ports and protocols (TCP or UDP) are randomly generated.

```
...
count src 110.10.179.196 tcp dst port 9196
count src 141.134.184.89 udp dst port 18489
count src 182.34.201.184 udp dst port 1184
count dst 156.54.137.236 tcp src port 37236
count dst 207.244.118.207 udp src port 18207
count src 106.75.133.10 tcp dst port 13310
count dst 191.6.91.22 udp src port 9122
count dst 83.209.148.98 udp src port 14898
...
```

The test results for the `rs-headers` ruleset are shown in Figure 5.5. This type of offloading yields even less benefit than in the `rs-ip` ruleset, with a maximum average reduction in the number of cycles of 2.9% for `specific` traffic in the case when every field is offloaded (*All* case). The offload of the protocol field (*Protocol* case) shows a small increase in the number of cycles (lower throughput) for both traffic types. Although it does not seem logical that there is a performance drop, it should be considered that the addition of metadata leads to an increase in the total packet size. In addition, the internal allocation of the required metadata to program memory could have a negative impact on the operation of the software filter.

Since both the `rs-ip` and `rs-headers` rulesets are offloaded using `metadata1`, the information from the packet being checked is not extracted from the packet during filtering, but is stored in the metadata at a predetermined location. An example of the implementation of such an offload method for the system used in this thesis is previously shown in Listing 4.4.



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.5: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-headers`.

Ruleset `rs-partoff`

Similar to `rs-headers`, in this ruleset, a portion of which is shown in Listing 5.3, all 1000 rules check the source / destination IP addresses, but these checks are considered p_N *patterns*. Additionally, each rule checks a p_O *pattern* of a randomly generated UDP port range. Therefore, the category of these rules is P_0. Technically, the UDP check should also be considered a separate *pattern*, but for simplicity this check has been merged with the port range check and can be considered a single *pattern*.

Offloading P_0 rules uses the `metadata2` metadata type. This metadata contains a bitmap with a non-match or match value (0 or 1) for an offloaded p_O *pattern* in a rule. In the software filter, this is implemented by checking the bitmap for each rule instead of checking the port range that would otherwise be checked at this point, as shown in Listing 5.4. As mentioned earlier, this C code is automatically generated as each ruleset is parsed and compiled into an executable program. Part of the efficiency of this software filter is achieved by using “hard-coded” values (such as IP address `0x6e0ab3c4` or protocol `17`), since the compiler can optimize

Listing 5.3: Excerpt from the ruleset `rs-partoff`. All IP addresses and port ranges are randomly generated.

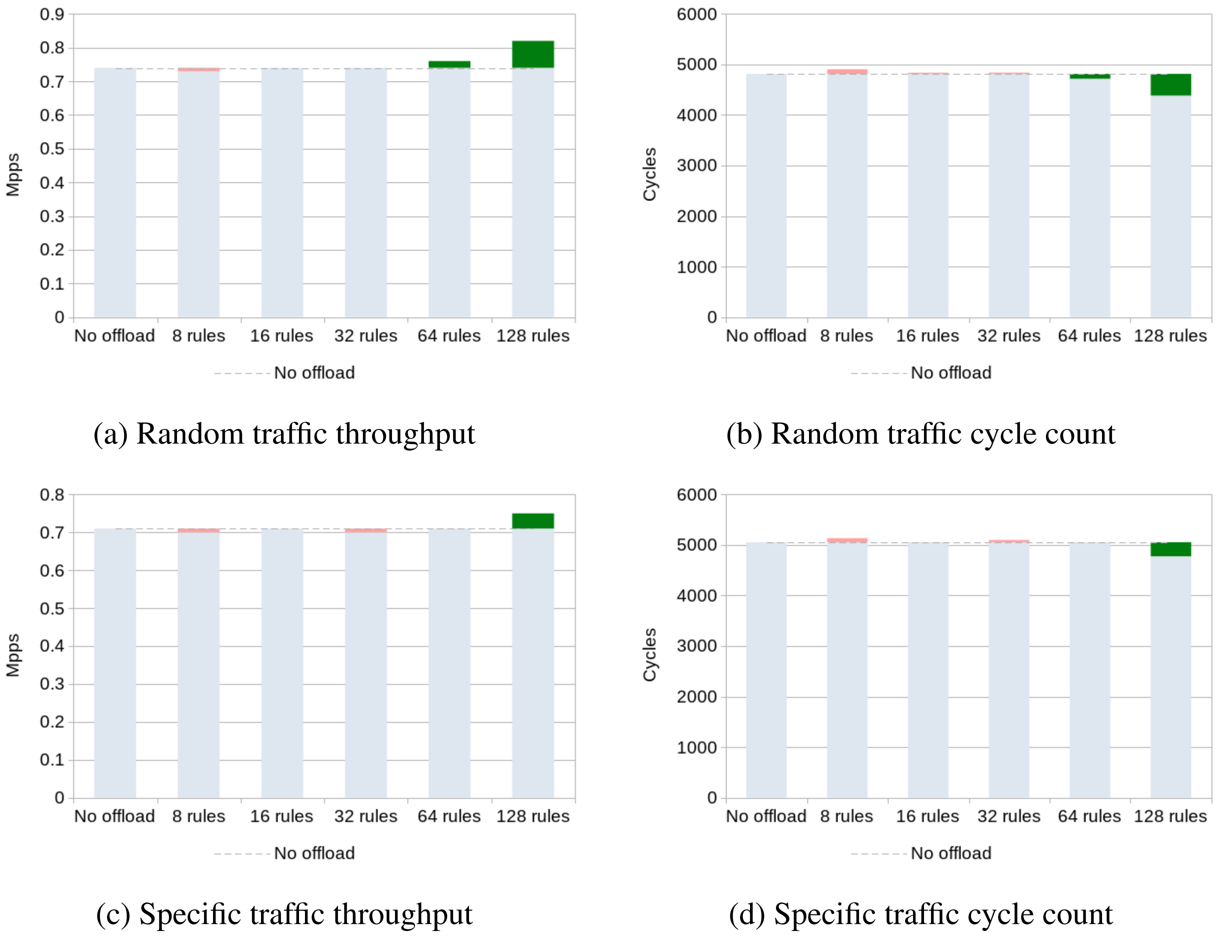
```
...
count src 110.10.179.196 udp dst port 19-919
count src 141.134.184.89 udp dst port 8489-18489
count src 182.34.201.184 udp dst port 1184-1184
count dst 156.54.137.236 udp dst port 7236-37236
count dst 207.244.118.207 udp dst port 8207-18207
count src 106.75.133.10 udp dst port 3310-13310
count dst 191.6.91.22 udp dst port 12-912
count dst 83.209.148.98 udp dst port 4898-14898
...
```

these constant values during compilation.

Listing 5.4: Software implementation of the `metadata2` offload.

```
/* Non-offloaded rule check */
if ((ip_src == 0x6e0ab3c4) && // if src IP is 110.10.179.196 AND
  (ip_p == 17 && // protocol is UDP AND
    ((t = dport) >= 19 && t <= 919)) // dport is between 19, 919
      COUNT(1); // count rule as matched

/* Offloaded rule check */
if ((ip_src == 0x6e0ab3c4) && // if src IP is 110.10.179.196 AND
  (meta->rulebits & 0x80) // metadata most significant bit is 1
    COUNT(1); // count rule as matched
```

The results are shown in Figure 5.6. For `specific` traffic the results show increases in the number of CPU cycles (up to 1.6% more) for smaller numbers of offloaded *patterns* (<64), while the 5.5% speedup is visible when 128 port ranges are offloaded. For `random` traffic, the results are similar, with a 2% speedup for 64, increasing to 8.9% for 128 offloaded *patterns*.



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.6: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-partoff`.

Ruleset `rs-fulloff`

The `rs-fulloff` ruleset is similar to the `rs-partoff` ruleset, but its 1000 (*counting*) rules only check port ranges, while IP addresses are not checked at all, as it can be seen in Listing 5.5. Since these port ranges are offloaded to hardware, they are considered p_O *patterns*. As the rules contain only p_O *patterns*, they fall into the *ONc* category.

Listing 5.5: Excerpt from the ruleset `rs-fulloff`. All port ranges are randomly generated.

```
...
count udp dst port 19-919
count udp dst port 8489-18489
count udp dst port 1184-1184
count udp dst port 7236-37236
count udp dst port 8207-18207
count udp dst port 3310-13310
count udp dst port 12-912
count udp dst port 4898-14898
...
```

Therefore, metadata3 is used to offload the rules. The implementation for metadata3 uses the same bitmap technique as for metadata2, but there are no additional checks for the same rule. An example of the implementation can be seen in Listing 5.6.

Listing 5.6: Software implementation of the metadata3 offload.

```
/* Non-offloaded rule check */
if (ip_p == 17 && // if protocol is UDP AND
  ((t = dport) >= 19 && t <= 919)) // dport is between 19, 919
    COUNT(1); // count rule as matched

/* Offloaded rule check */
if (meta->rulebits & 0x80) // if metadata most significant bit is 1
  COUNT(1); // count rule as matched
```

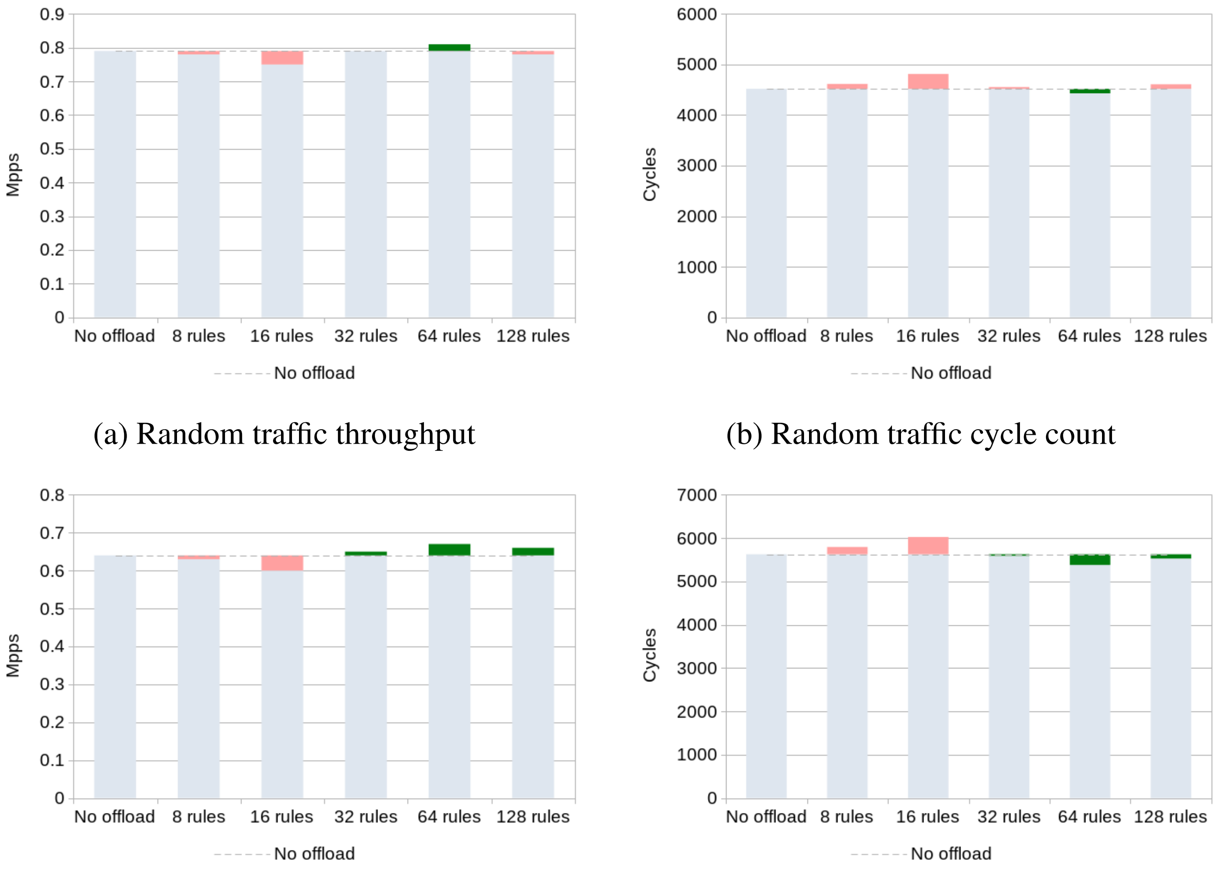
The results for this offload type can be seen in Figure 5.7. The results for random traffic show the performance degradation regardless of the number of offloaded rules (up to 6.5% increase in cycles) except for 64 offloaded rules (2% decrease in cycles). The results for specific traffic show degradation for 8 and 16 offloaded rules (up to 7.1% increase in cycles) and improvements for the rest (up to 4.4% for 64 rules).

Protection against DDoS attacks is the priority of this filter. So, although there is performance degradation for random traffic, offloaded filtering of specific traffic shows improvement over non-offloaded filtering. The filter can be configured to adapt to different types of traffic by enabling / disabling hardware offload as needed.

Since there are no IP address checks in this ruleset, the specific traffic has been modified to use 90% of the ports from any port range in the ruleset and 10% of the random ports. This specification of traffic also applies to other rulesets with port-only checks, while rulesets with IP address checks and port checks use the combination of both values when generating traffic.

Ruleset rs-skipa

The rs-skipa ruleset is similar to the rs-fulloff ruleset, but belongs to the second group of rulesets because it uses *terminating* rules. Of its 1000 rules, some are not checked in the software, but are marked as offloaded. The software only checks if they match the rule number read from the metadata. Part of the ruleset is shown in Listing 5.7. All port ranges of the 128 offloada rules are sequential (from 0-63 to 8128-8191) so that traffic matches each rule with equal probability. All other port ranges in the *count* rules are randomly generated, as in the rs-fulloff ruleset. Rules marked with offloada consist only of one *Fully offloadable pattern* (p_O) and terminate with the *Accept action*, which increments the counter, putting these rules in the *OAc* category.



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.7: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-fulloff`.

Listing 5.7: Excerpt from the ruleset `rs-skipa`.

```
...
count udp dst port 7203-37203
offloada udp dst port 1664-1727
count udp dst port 245-10245
count udp dst port 12-412
count udp dst port 19-419
offloada udp dst port 1728-1791
...
```

The *OAc* category uses the `metadata4` type of offload. An example of the software implementation is given in Listing 5.8. When this type of offload is enabled, the rules marked with the `offloada` keyword are not checked, as they normally would be when the C code is generated. Instead, the program checks the rule number from the metadata. For each `offloada` rule that is not offloaded to hardware, the usual software check is performed.

Listing 5.8: Software implementation of metadata4 offload.

```
#if OFFLOAD & META4 // metadata4 is active
    if (meta->skip == 11) // if metadata is offloaded rule #11
#else // metadata4 is not active
    if (ip_p == 17 && // if protocol is UDP AND
      ((t = th_dport) >= 12 && t <= 912)) port is between 12, 912
#endif
        ACCEPT(18); // count and accept rule as matched
```

As explained in previous chapters, metadata4 is created for cases where the packet must be dropped / forwarded if a rule is matched and all subsequent rules are not checked at all. This test is different from the previous ones because such packets are forwarded. This can increase the overall throughput as all other rules after the matching one do not need to be checked, saving CPU cycles.

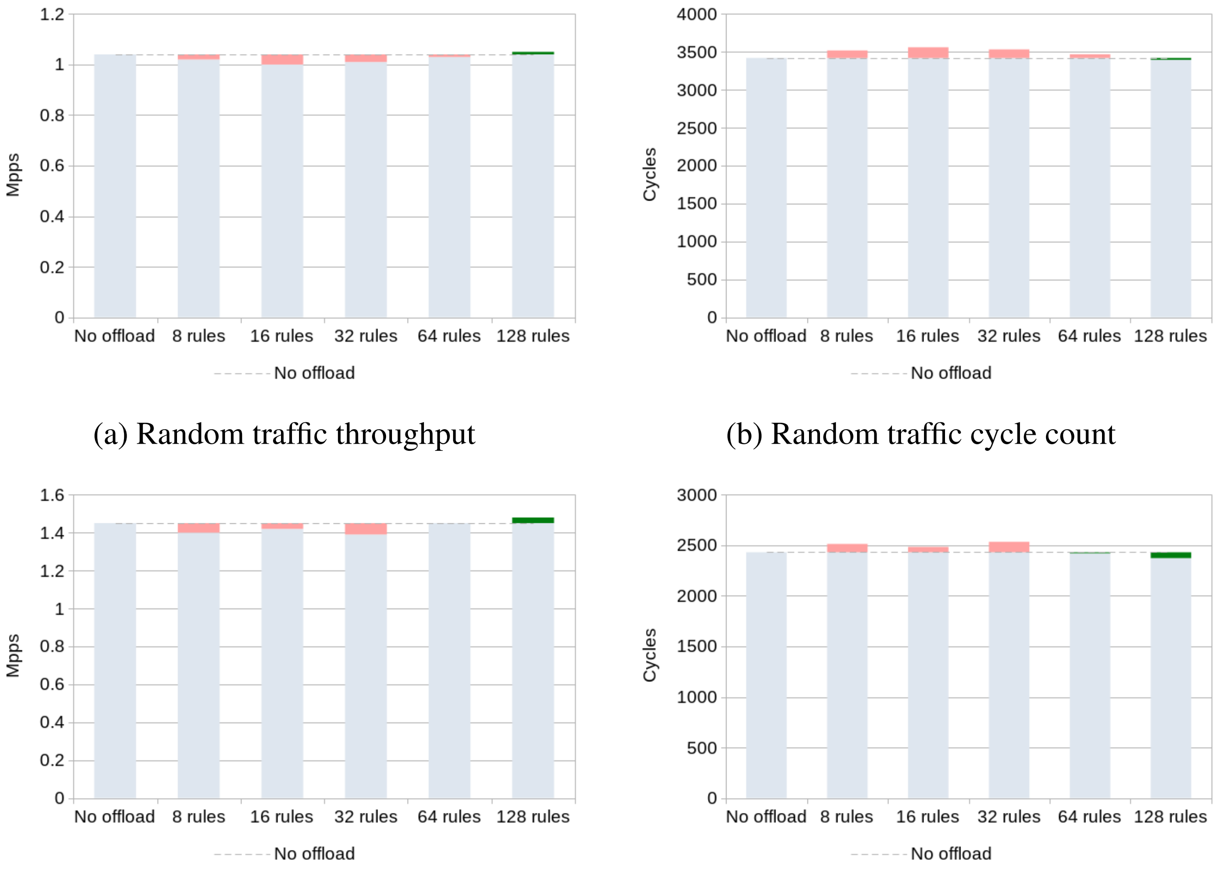
This type of metadata could be implemented in the same way as metadata3, but this would mean that each metadata entry would have to have two bits (for a total of three combinations — no match, match *Accept* and match *Deny*). This is because multiple rules can be matched at the same time, but only one *action* is performed in each case.

The results in Figure 5.8 show that the number of offloaded rules affects performance. For specific traffic, when the number of rules is smaller (8, 16, and 32 offloaded rules out of a total of 1000) the performance decreases (the number of cycles increases by about 2–4%). For 64 offloaded rules, there is no significant change in performance, but for more rules (128), performance increases (the number of cycles decreases by about 2.4%). For random traffic, this type of offloading also leads to worse performance, with only 128 offloaded rules having a somewhat positive impact on performance (the number of cycles decreases by 0.7%). The results show that the ratio of offloaded to total rules must be as large as possible to achieve a positive performance change. However, due to hardware limitations, there is a limit to this ratio, which is discussed in the hardware section of this chapter.

Alternatively, metadata4 offload can be implemented differently if such rules are used only at the beginning, before all other rules in the ruleset, i.e., only if there are no software or hybrid rules before them (as shown in Listing 5.9).

In this way, the software does not need to check each offloaded rule individually, but only needs to perform one check before proceeding with the rest of the rules: if the rule matches in hardware, the offload metadata is non-zero and the software should simply perform the appropriate action for that rule, as shown in Listing 5.10. If the offloaded rules are non-terminating rules, there should also be multiple metadata fields and this code should iterate through them all.

The results for this ruleset are shown in Figure 5.9. For specific traffic, this type of



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.8: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-skipa`.

Listing 5.9: Excerpt from the alternative ruleset `rs-skipa_alt`. All port ranges are the same as in the original ruleset, only their order is changed.

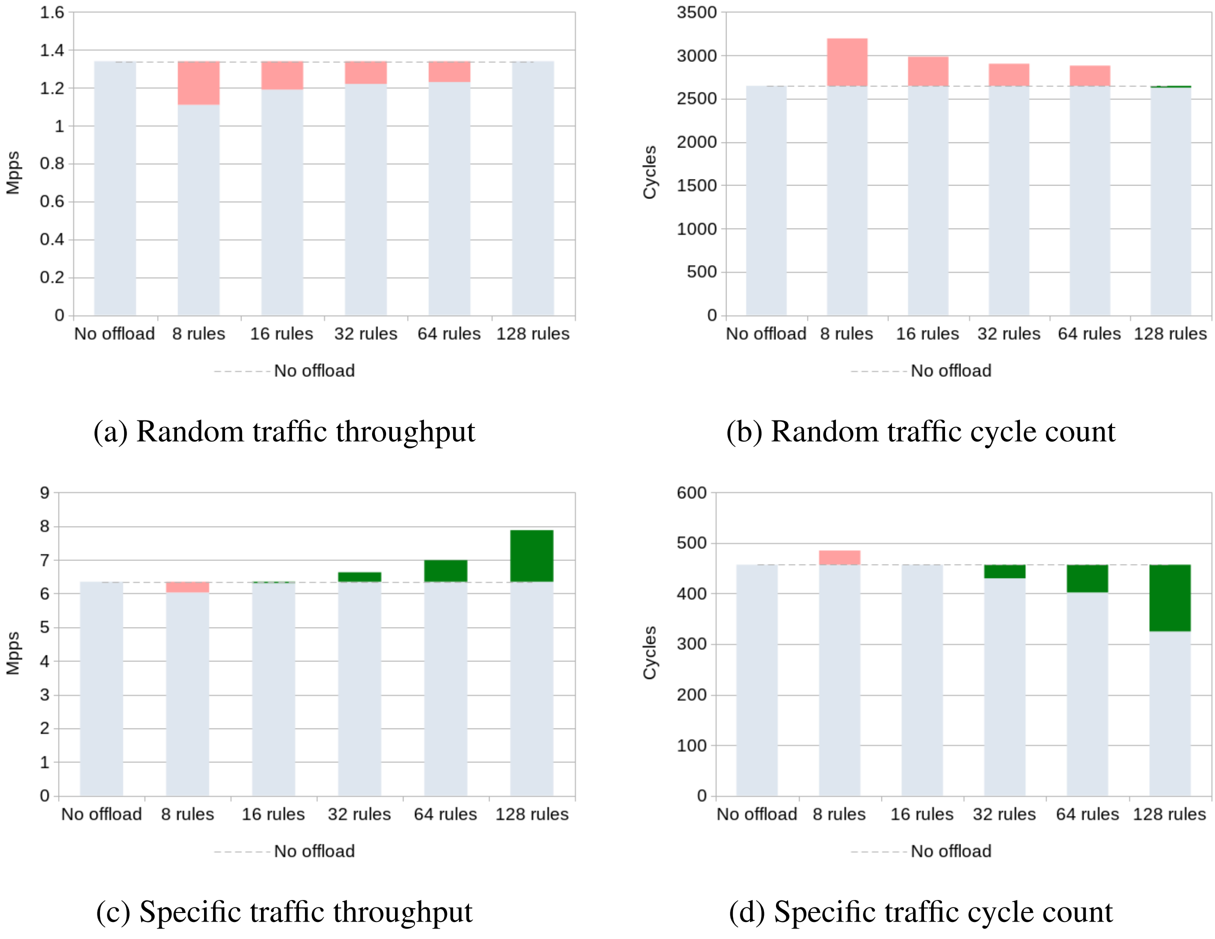
```
offloada udp dst port 0-63
offloada udp dst port 64-127
offloada udp dst port 128-191
offloada udp dst port 192-255
offloada udp dst port 256-319
...
count udp dst port 19-919
count udp dst port 8489-18489
count udp dst port 1184-1184
count udp dst port 7236-37236
count udp dst port 4898-14898
...
```

performance improvement from offloading is more pronounced than for the original ruleset. The performance increase is visible for as few as 16 rules and increases to 28.9% for 128 offloaded rules. This is because the generated filter has fewer “if” conditions, which makes

Listing 5.10: Alternative software implementation of `metadata4` offload.

```
    if (meta->skip) // if the matched rule is non-zero
        ACCEPT(meta->skip); // increment its counter and terminate
```

CPU's job easier and optimizes it by generating fewer possible branches when compiling the C code (essentially, the ruleset behaves as if it has fewer rules in the ruleset). Also, most of the traffic is concentrated on the offloaded rules, so this has a greater impact than in the `random` traffic scenario (only 0.8% decrease in cycle count with 128 offloaded rules).



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.9: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-skipa_alt` (first rules offloaded).

Ruleset `rs-lpm`

The `rs-lpm` ruleset differs from previous rulesets in that it uses prebuilt tables (lists) populated with various IP addresses and subnets. The ruleset consists of 9 rules using a total of 9 tables, and is shown in full in Listing 5.11. Each rule consists of one *Partially offloadable pattern* (p_P), which makes these rules category P_1.

Listing 5.11: Ruleset `rs-lpm`.

```
include gl2_city.cfg    # 3,075,452 prefixes
include blocklist_3.cfg # 1,113,394 prefixes
include gl2_asn.cfg     # 430,976 prefixes
include gl2_country.cfg # 338,012 prefixes
include allowlist_3.cfg # 264,364 prefixes
include allowlist_2.cfg # 201,478 prefixes
include blocklist_1.cfg # 177,936 prefixes
include allowlist_1.cfg # 82,157 prefixes
include blocklist_2.cfg # 11,213 prefixes

count src table allowlist_1 any             # rule #1
count src table allowlist_2 any             # rule #2
count src table allowlist_3 any             # rule #3
count src table block_3 any                 # rule #4
count src table gl2_country HR,US,JP,CN,SR  # rule #5
count src table gl2_city 1880252            # rule #6
count src table block_2 any                 # rule #7
count src table block_1 any                 # rule #8
count src table gl2_asn 1221                # rule #9
```

Ranges of IP addresses allocated to specified cities, countries, and AS's are listed in tables `gl2_city`, `gl2_country`, and `gl2_asn`. Each entry in the table is labeled with a code belonging to a certain city, country, or AS and can be used to create rules such as #5, #6, or #9. The blocklists contain IP addresses taken from various publicly available collectors of malicious and dangerous IP addresses. The allowlists were created by monitoring the network traffic of the Faculty of Electrical Engineering and Computing, Department of Telecommunications for one month. The traffic payload was ignored, only the source/destination IP addresses were recorded. For the purposes of this thesis, only public IP addresses are listed and considered secure. Local IP addresses were removed as traffic was observed behind a NAT. Blocklists and allowlists are intentionally divided into 3 tables of different sizes to further complicate the work of the filter.

With such tables and LPM it is possible to filter a much larger number of network prefixes. In this case, all 9 tables contain a total of 5.6 million prefixes.

Like the `rs-ip` and `rs-headers` rulesets, this ruleset uses the same type of offload, which only "helps" the software filter by attaching useful data to the packet (`metadata1`). However, because the `rs-lpm` ruleset uses LPM lookups, this software implementation is different, as shown in Listing 5.12.

By using `if` preprocessor directives while compiling the C code, the implementation of this type of offload selects functionality based on whether `metadata1` is enabled. If non-offloaded filtering is used, the function performs the "regular" LPM lookup, as explained in Section 4.2.3. Otherwise, part of the function is skipped and the precomputed `direct_entry` structure is forwarded from the metadata. The rest of the code (i.e., the *range_lookup* function) is called in

Listing 5.12: Software implementation of `metadata1` offload for LPM tables.

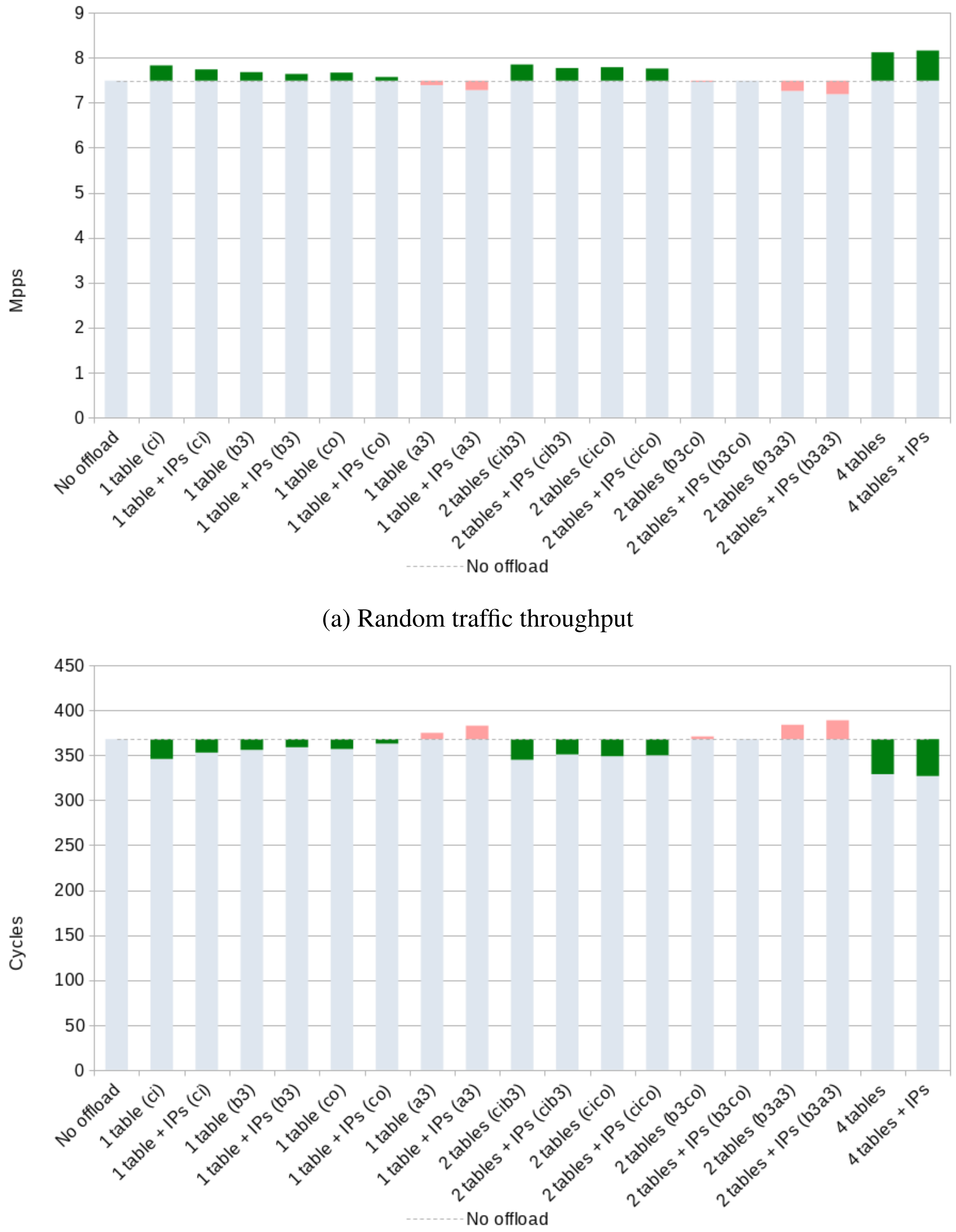
```
#if OFFLOAD == META1
static int
tbl_lookup(uint32_t dst, struct direct_entry de, void *rp)
{
#else
static int
tbl_lookup(uint32_t dst, uint32_t dxr_d, void *dp, void *rp)
{
    int di;
    uint16_t *dt = dp;
    struct direct_entry *xt = (void *) &dt[1 << dxr_d];
    struct direct_entry de;

    di = (dt[dst >> (32 - dxr_d)] << (20 - dxr_d)) + ((dst >> 12) &
      (0xffffffff >> (32 - (20 - dxr_d))));
    de = xt[di];
#end
    if (predict_true(de.fragments == 1014))
      return (de.base);

    return (range_lookup(dst, de, rp));
}
```

both cases.

The goal of this test was to measure the performance by offloading different LPM tables, as well as the performance of a different number of LPM tables. The results in Figure 5.10 and Figure 5.11 show a comparison of filtering with different combinations of offloaded LPM tables for `random` and `specific` traffic. To observe the effect of offloading additional metadata of the same type (`metadata1`) when filtering with the same ruleset, another batch of tests was performed. In addition to offloading LPM tables, the metadata also included the source IP address of the packet.

Offloading `blocklist_3` (`b3`) and `gl2_country` (`co`) tables has the largest impact because these tables cover the largest number of packet prefixes. When the impact of the offloaded tables is smaller, the performance drops about 2% below the baseline performance (e.g., for `allowlist_3` — `a3`). This minor drop proves that offloading does not significantly affect filtering performance when processing non-critical traffic. The highest performance gains are seen when using four-table metadata (approximately 10.5% fewer cycles are used for both types of traffic) and with source IP address (the number of cycles decreases by 11.6% for `specific` and 11.1% for `random` traffic).

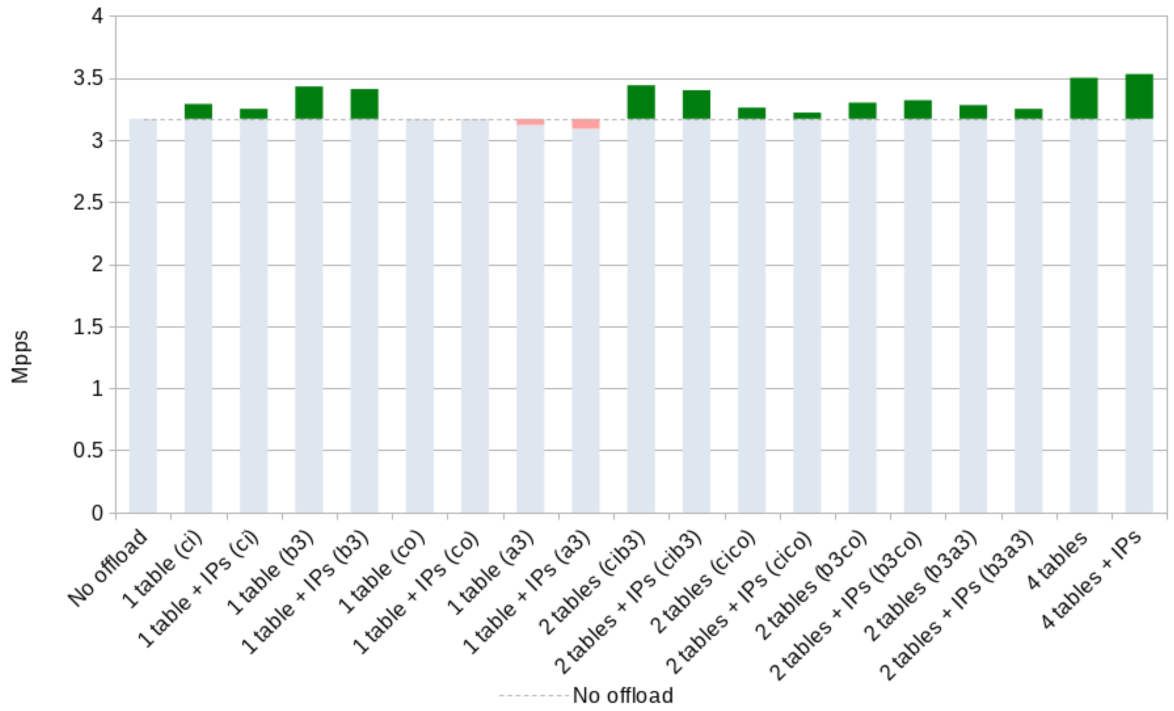
The performance of the filter with additional source IP addresses in the metadata is higher in one case with two LPM tables (`b3co`) and in the case with four LPM tables. In all other cases, the additional metadata increased the performance of the filter by about 2% less than the filtering when offloading only LPM tables.



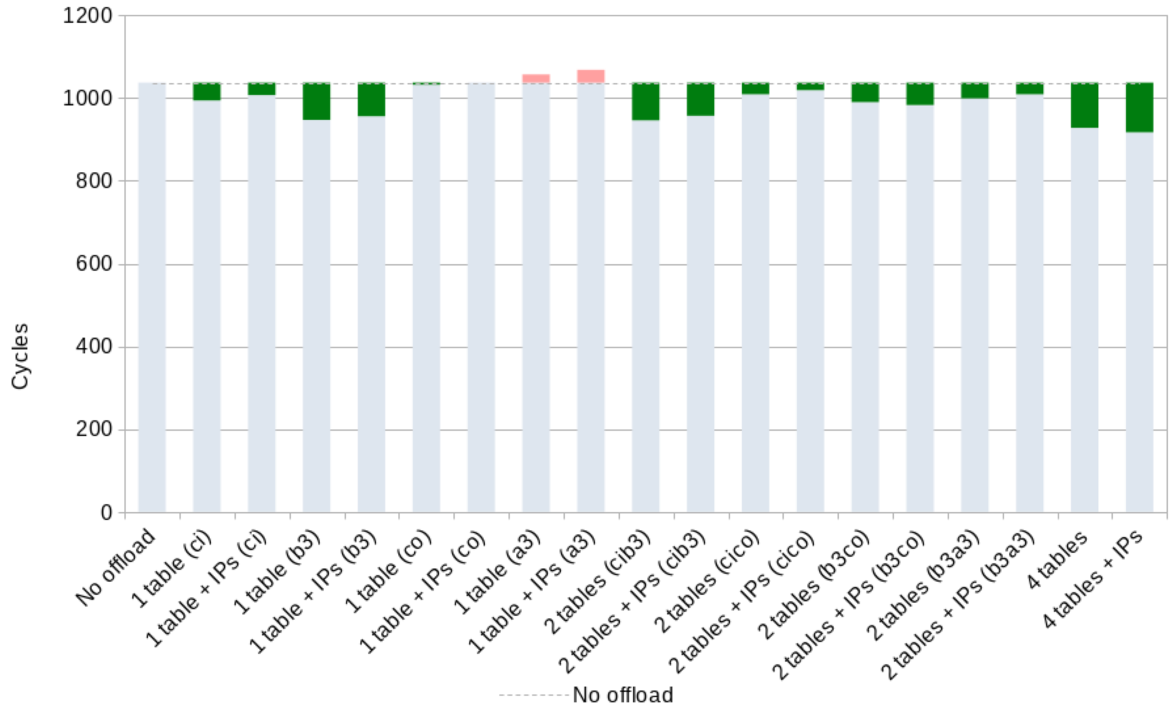
(a) Random traffic throughput

(b) Random traffic cycle count

Figure 5.10: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpm` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.11: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpm` for `specific` traffic.

Ruleset `rs-lpmpart`

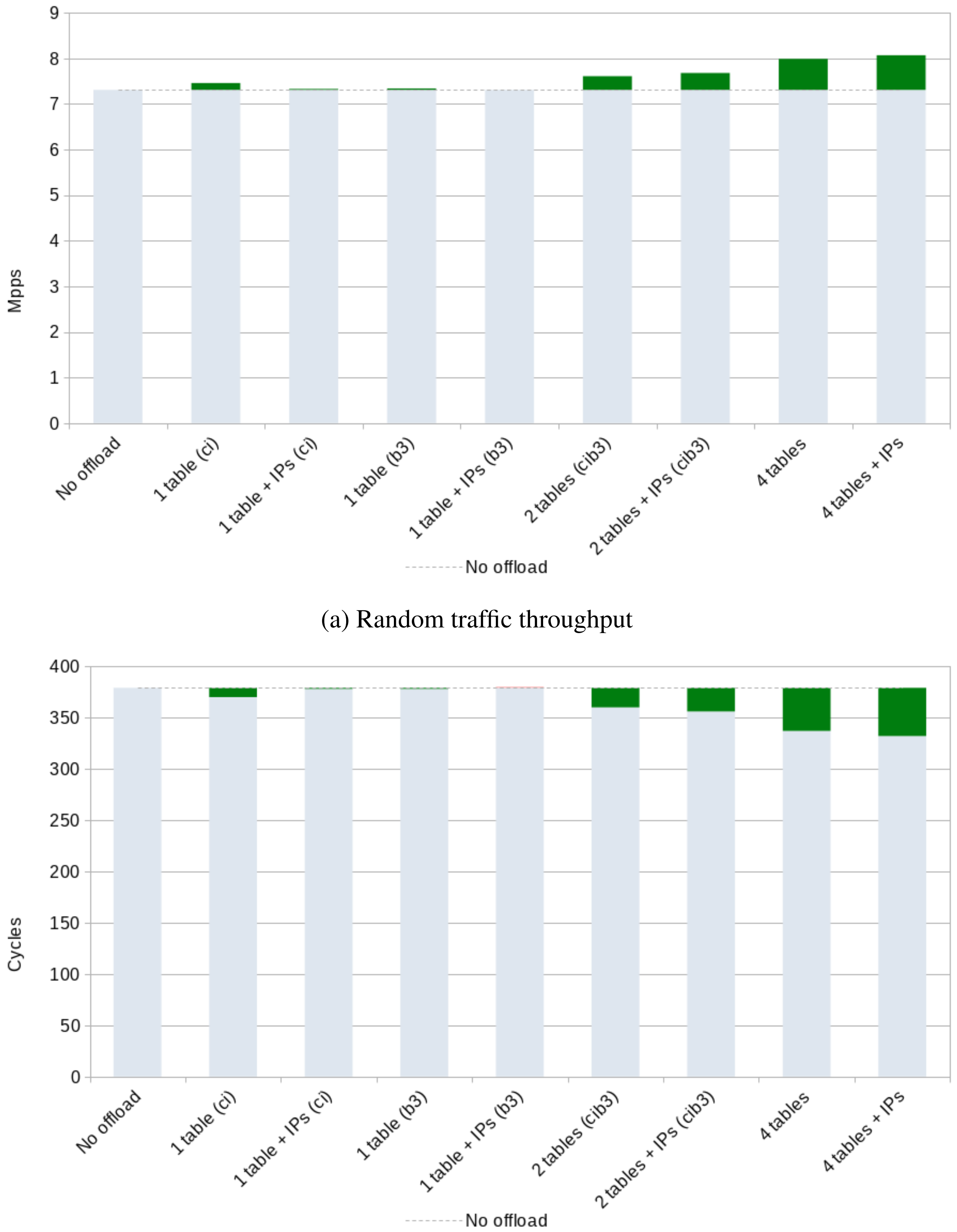
The `rs-lpmpart` ruleset, shown in Listing 5.13 is similar to the `rs-lpm` ruleset, but 8 (out of 9) rules also have a port range check included in the rule. This places these rules in the P_2 category and serves as a way to test the combination of `metadata1` and `metadata2`. The tests were done only for tables that showed performance improvement in the `rs-lpm` tests.

Listing 5.13: Excerpt from the ruleset `rs-lpmpart`. Port range checks are taken from the `rs-partoff` ruleset.

```
include gl2_city.cfg    # 3,075,452 prefixes
include blocklist_3.cfg # 1,113,394 prefixes
include gl2_asn.cfg     # 430,976 prefixes
include gl2_country.cfg # 338,012 prefixes
include allowlist_3.cfg # 264,364 prefixes
include allowlist_2.cfg # 201,478 prefixes
include blocklist_1.cfg # 177,936 prefixes
include allowlist_1.cfg # 82,157 prefixes
include blocklist_2.cfg # 11,213 prefixes

count src table allowlist_1 any udp dst port 9658-19658
count src table allowlist_2 any udp dst port 663-1663
count src table allowlist_3 any udp dst port 10-310
count src table block_3 LV3 udp dst port 376-10376
count src table gl2_country HR,US,JP,CN,SR udp dst port 1202-31202
count src table gl2_city 1880252 udp dst port 201-6201
count src table block_2 any udp dst port 6213-16213
count src table block_1 any udp dst port 6202-46202
count src table gl2_asn 1221
```

The results for this ruleset, shown in Figure 5.12 and Figure 5.13, are similar for both `random` and `specific` traffic. It shows that using multiple tables for offloading gives better results. For `random` traffic, using one LPM table (`gl2_city` — `ci`) resulted in a 2.4% performance improvement. Using two LPM tables (combination of `gl2_city` and `blocklist_3` — `cib3`) increased performance by 5%. Using four LPM tables increased performance by 11.1%. Using one LPM table (`blocklist_3` — `b3`) increased performance by 5.2% for `specific` traffic. Using two LPM tables (combination of `gl2_city` and `blocklist_3` — `cib3`) increased performance by 7.5%. The use of four LPM tables increased the performance by 9.9%.

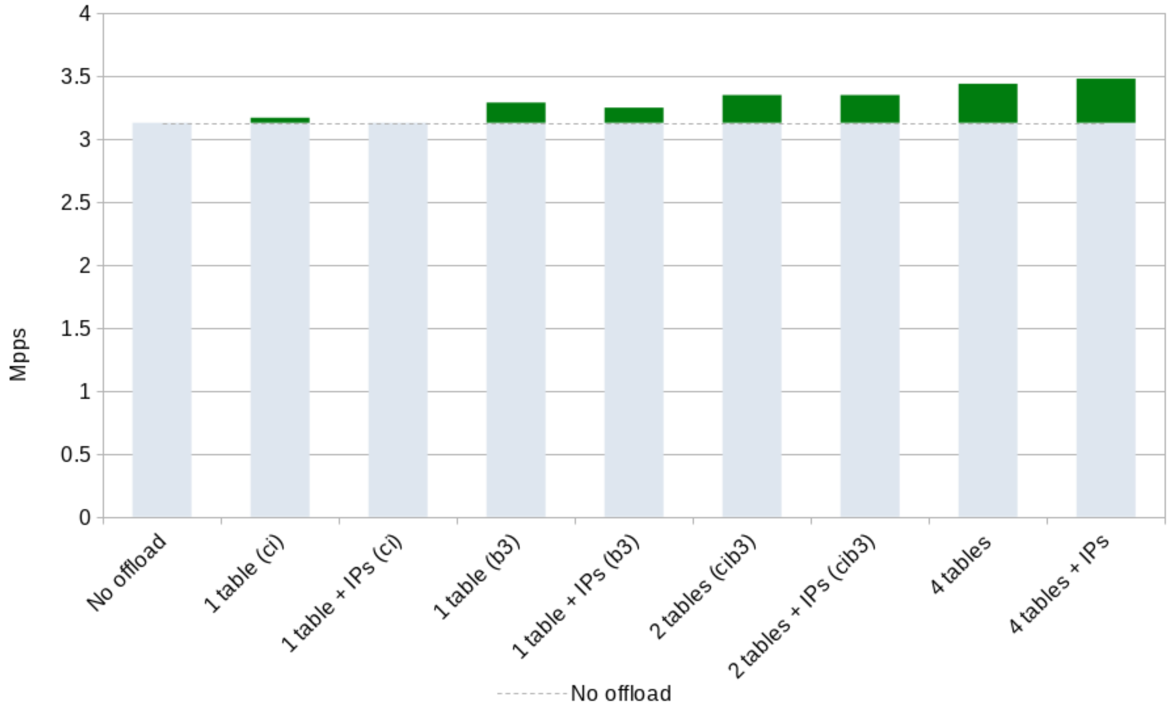
Using additional port range offloading further improves performance when using four LPM tables for both traffic types (a difference in the number of cycles of 1.5–2%). When offloading one table, the results are worse when using port range offloading (a difference in the number of cycles of 1–2%), and when offloading two tables, the results are similar for both cases. Using four tables is the best case and increases performance for both traffic types (12.4% fewer cycles without additional port range offloading for `random` traffic and 11.2% for `specific` traffic).



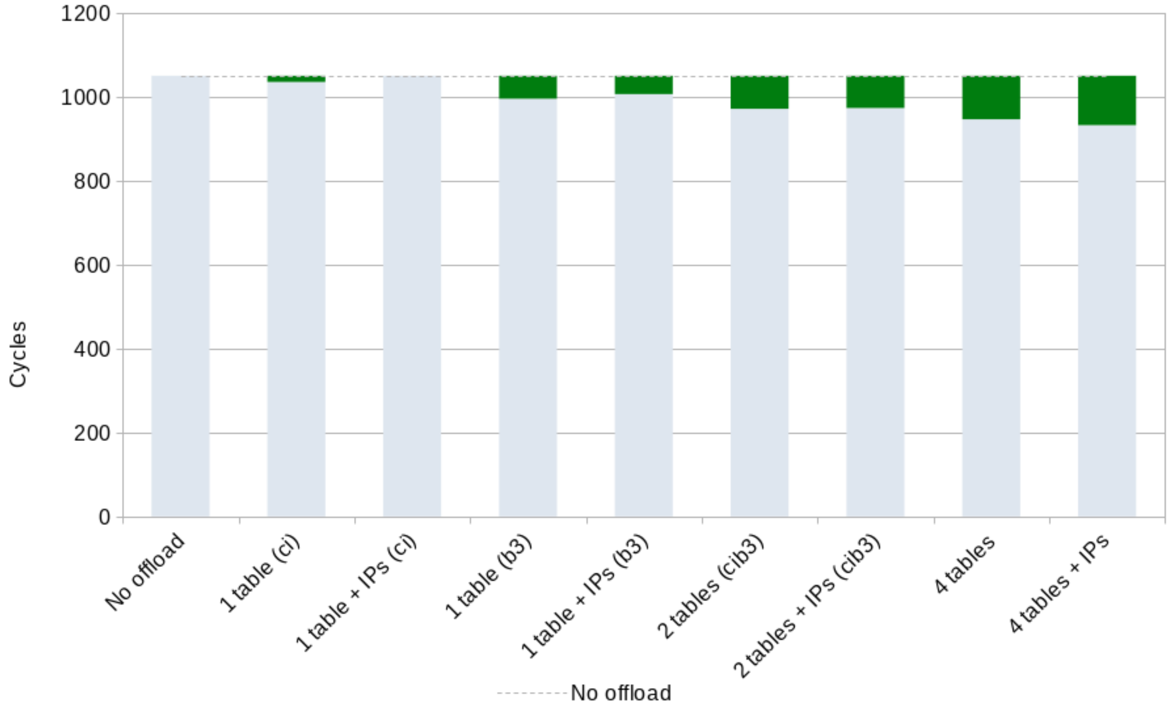
(a) Random traffic throughput

(b) Random traffic cycle count

Figure 5.12: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmpart` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.13: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmpart` for `specific` traffic.

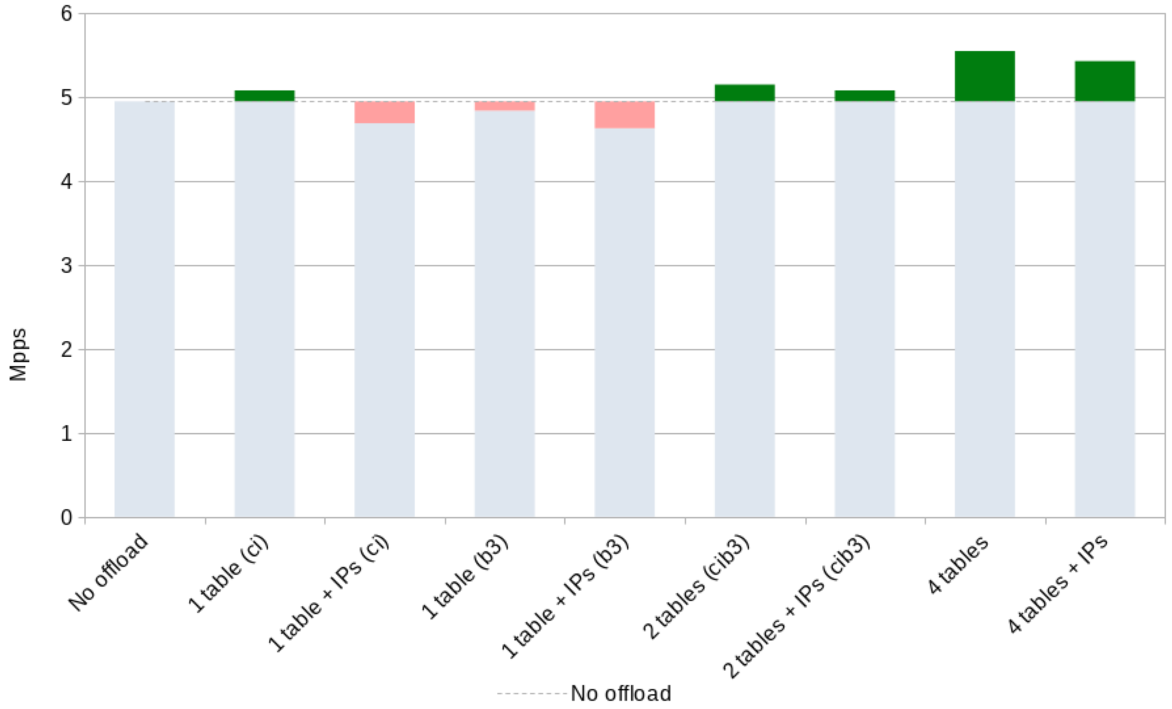
Ruleset `rs-lpmfull`

The `rs-lpmfull` ruleset is used to test a combination of `metadata1` and `metadata3` offloading because it is a combination of rules from the P_1 and *ONc* categories. It consists out of 128 fully offloadable *counting* rules (port ranges), with all 9 rules from `rs-lpm` randomly inserted into the ruleset, as shown in Listing 5.14. The tests were done only for tables that showed performance improvement in the `rs-lpm` tests.

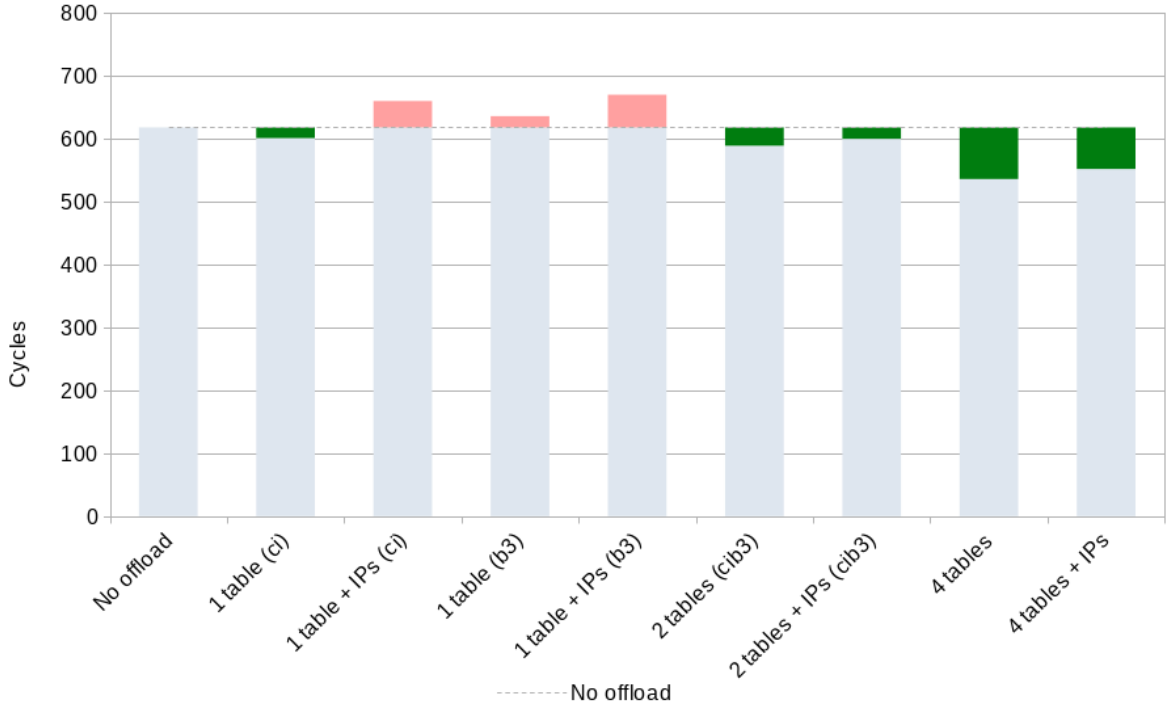
Listing 5.14: Excerpt from the ruleset `rs-lpmfull`. Port range checks are taken from the `rs-partoff` ruleset.

```
...
count udp dst port 5568-5631
count udp dst port 5632-5695
count udp dst port 5696-5759
count src table gl2_country HR,US,JP,CN,SR
count udp dst port 5760-5823
count udp dst port 5824-5887
count udp dst port 5888-5951
...
```

The results for this ruleset, shown in Figure 5.14 and Figure 5.15, are similar for both `random` and `specific` traffic. Using more tables for offloading gives better results in each test, but using additional port range offloading results in performance degradation (a difference in cycles of 1–6%) in each case compared to offloading by using only LPM tables. Using four tables is the best case as it increases performance for both traffic types (13.3% fewer cycles without additional port range offloading for `random` traffic and 12.3% for `specific` traffic).

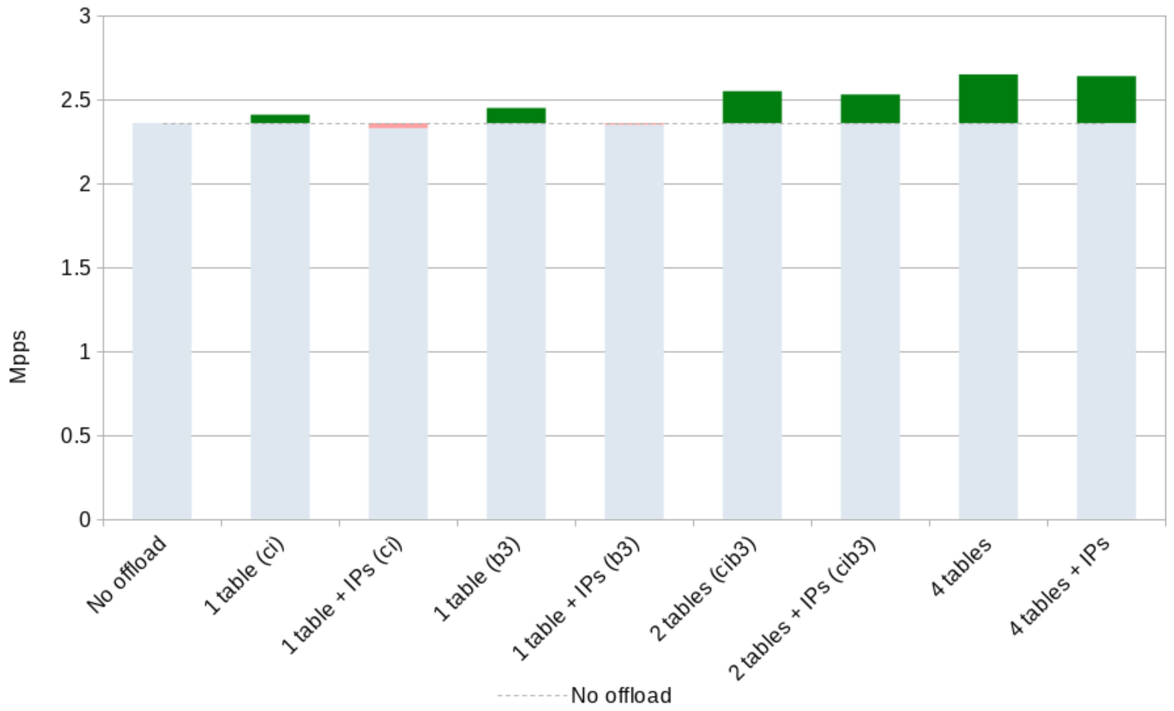


(a) Random traffic throughput

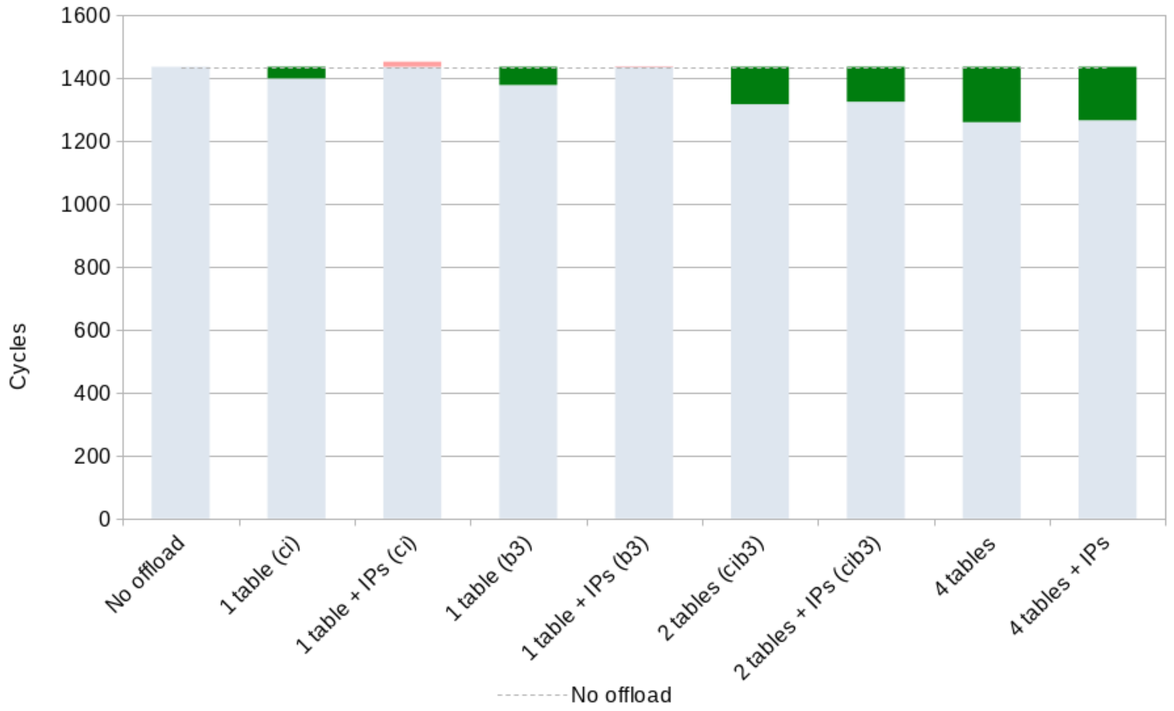


(b) Random traffic cycle count

Figure 5.14: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmfull` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.15: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmfull` for `specific` traffic.

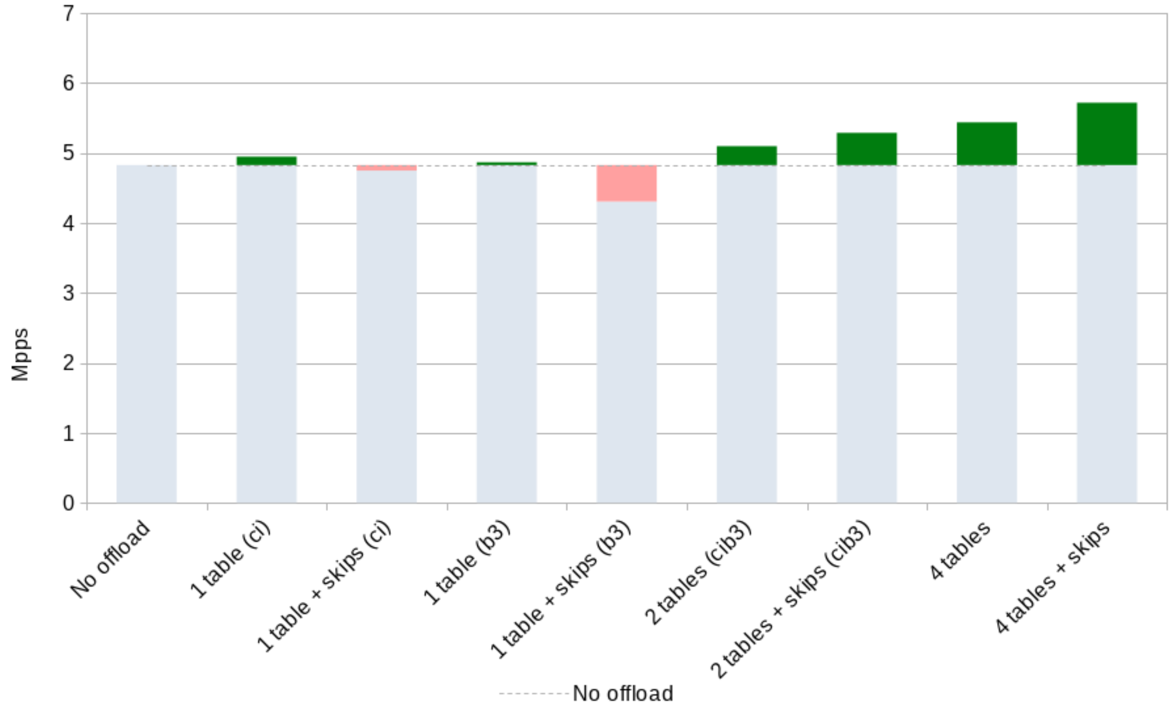
Ruleset `rs-lpmskipa`

This ruleset is a combination of the 128 `offloada` rules used previously and all 9 rules from the `rs-lpm` ruleset inserted randomly into it, as shown in Listing 5.15. Of the 128 rules used, only 16 were offloaded — to represent a more realistic case due to hardware limitations. This ruleset shows the results of filtering by combining `metadata1` and `metadata4`, since it consists of rules from categories P_1 and *OAc*.

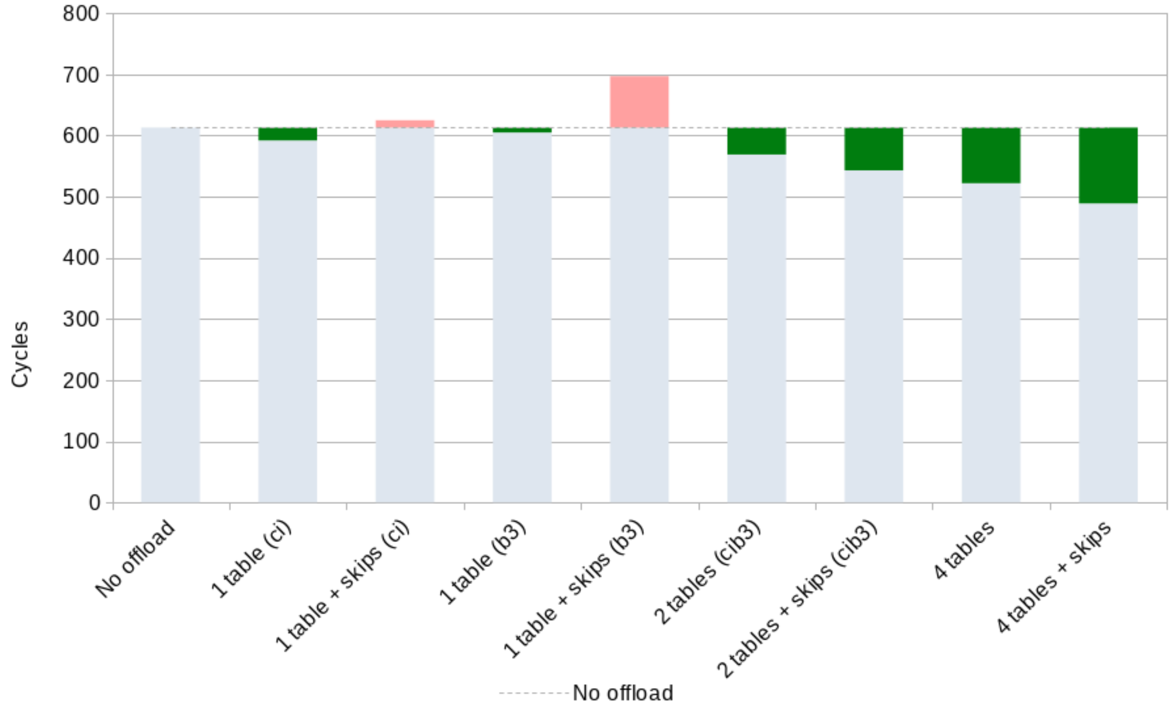
Listing 5.15: Excerpt from the ruleset `rs-lpmskipa`.

```
...
offloada udp dst port 768-831
offloada udp dst port 832-895
offloada udp dst port 896-959
offloada udp dst port 960-1023
count src table allowlist_1 any
offloada udp dst port 1024-1087
offloada udp dst port 1088-1151
offloada udp dst port 1152-1215
offloada udp dst port 1216-1279
...
```

The overall throughput of this ruleset is slightly higher compared to `rs-lpm` for `specific` traffic, since these additional *terminating* 128 port range checks cause the filter to skip most of the LPM checks in the ruleset. For this reason, the overall performance improvement depends on the combination of both offloads, with the impact of the `metadata4` offload being slightly stronger. The results in Figure 5.16 and Figure 5.17 show lower performance when only one table is offloaded, with the number of cycles increasing by up to 5%.

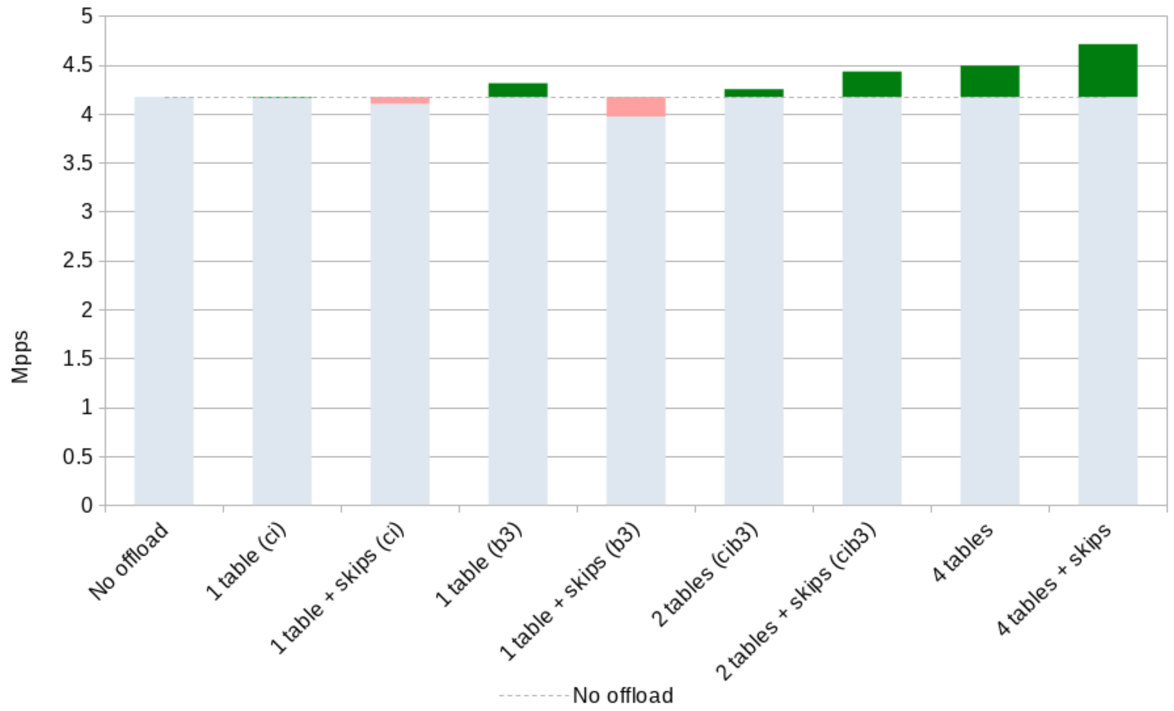


(a) Random traffic throughput

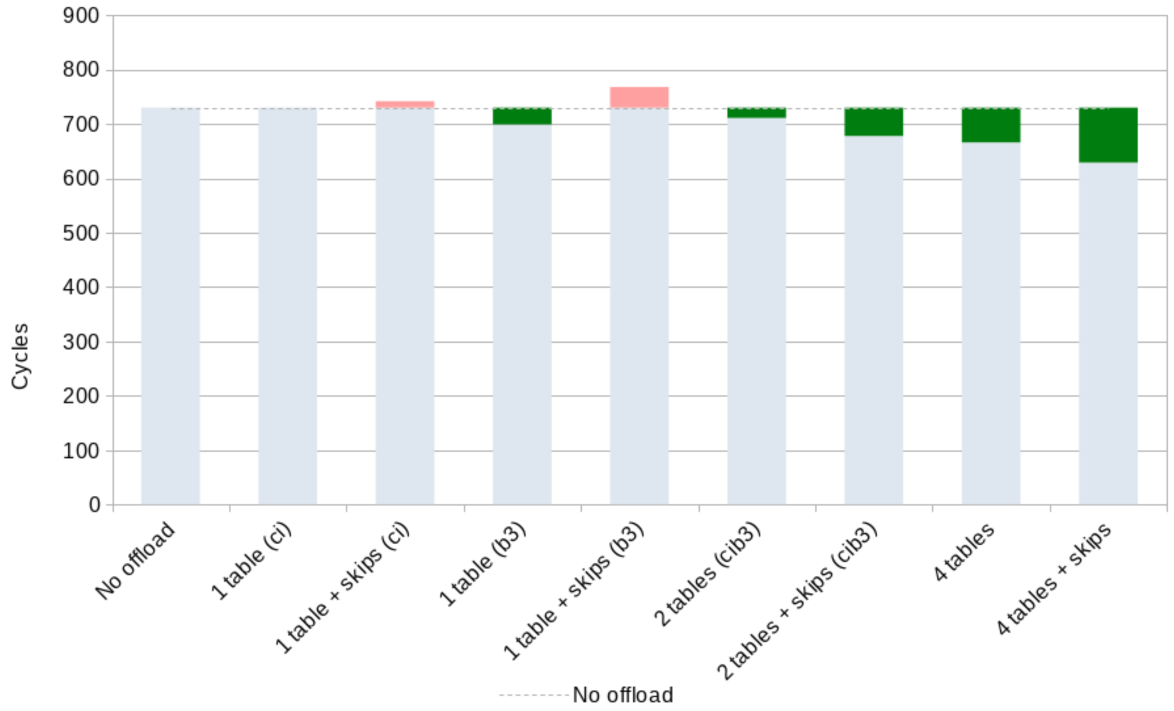


(b) Random traffic cycle count

Figure 5.16: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmskipa` for `random` traffic.



(a) Specific traffic throughput



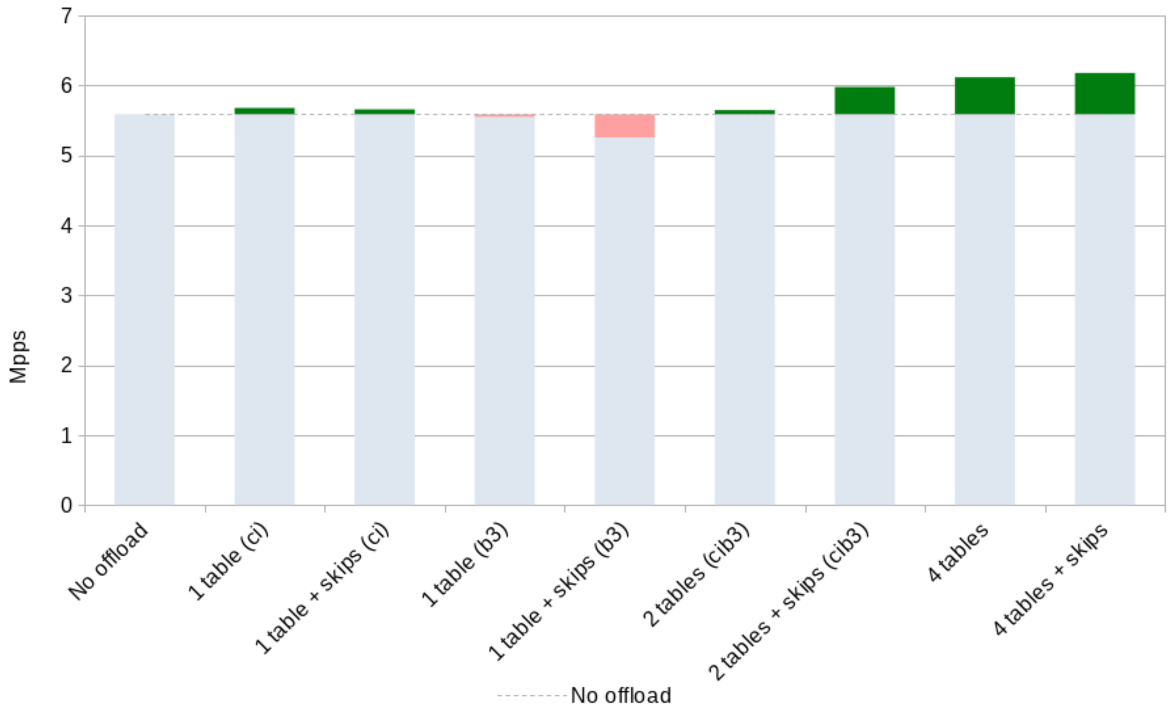
(b) Specific traffic cycle count

Figure 5.17: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmskipa` for `specific` traffic.

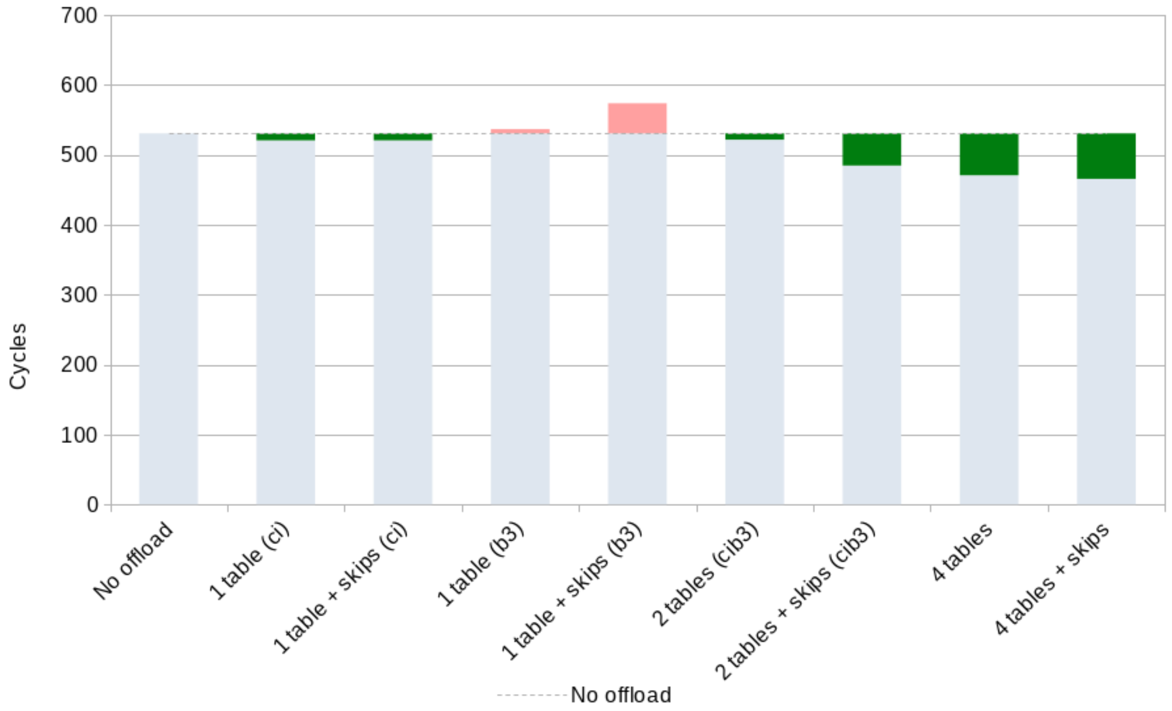
On the other hand, the throughput of the `random` traffic is lower compared to the `rs-lpm` ruleset because the total number of rules is increased. The same Figure 5.16 and Figure 5.17 show the performance increase for two tables (the cycle count decreases by 7% without and by 11.5% with additional offloaded rules) and for four tables (the cycle count decreases by 15% without and by 20% with additional offloaded rules).

As with the `rs-skipa` ruleset, the results would be different if all rules offloaded in hardware were placed at the beginning of the ruleset and the alternative software implementation was used, as shown in Figure 5.18 and Figure 5.19.

The performance increase for the `specific` traffic is lower with four offloaded tables than the one with additional 16 offloadable rules (the cycle count decreases by 2% without and 6.7% with additional metadata). Similarly for `random` traffic (cycle count decreases by 11% without, 12.2% with additional metadata). It is important to note that the total throughput for both `specific` and `random` traffic in this scenario is much higher than in the `rs-lpmskipa` ruleset. Comparing the performance gains for these two cases would not be fair, especially for `specific` traffic, where the first 128 rules have much more weight than the LPM rules, just as in the `rs-skipa` example.

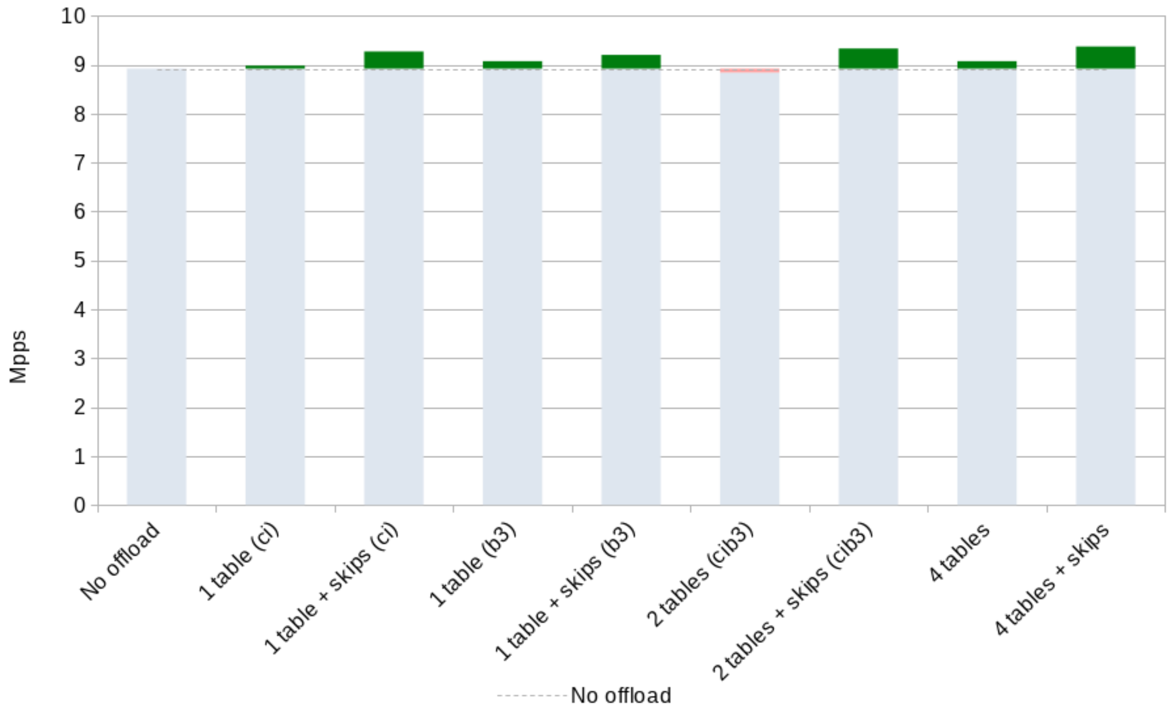


(a) Random traffic throughput

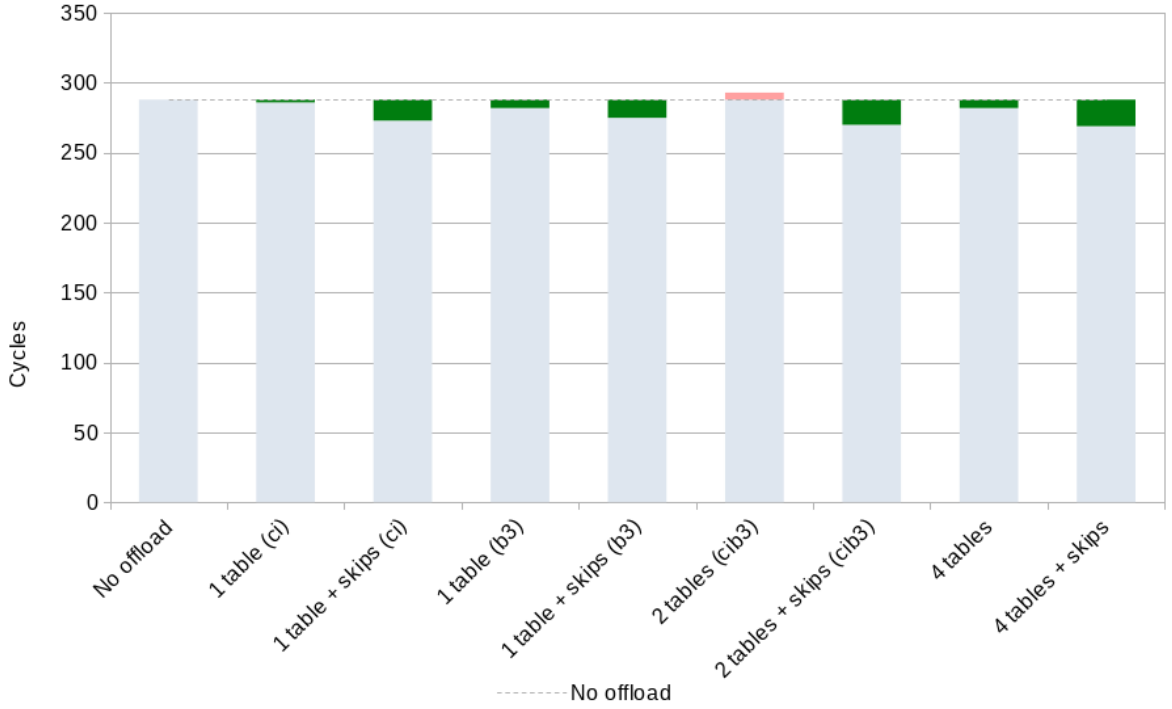


(b) Random traffic cycle count

Figure 5.18: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmskipa` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.19: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmskipa` for `specific` traffic.

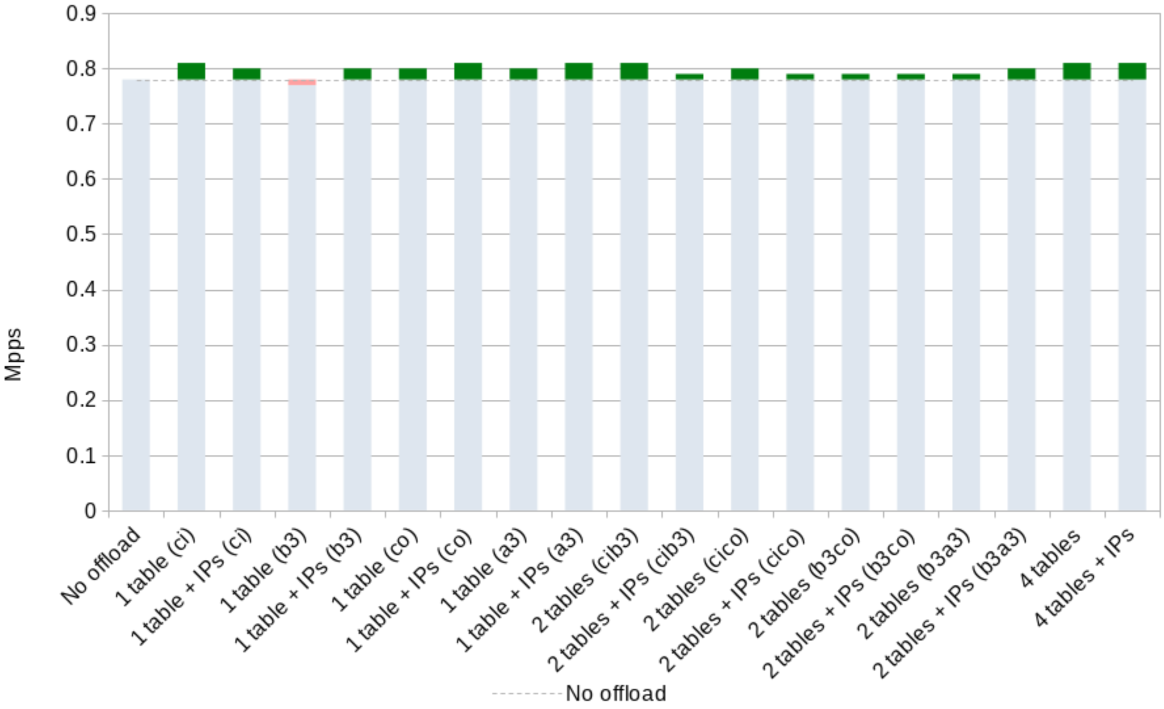
Ruleset `rs-mix`

To show the effects of a large number of rules even on the ruleset with the highest throughput, the `rs-mix` ruleset was created, as partially shown in Listing 5.16. It is a combination of rules from the `rs-lpm` ruleset, randomly inserted into the `rs-ip` ruleset so that all rules belong to the P_1 category. The total measured throughput was about 0.6 Mpps for `specific` and 0.8 Mpps for `random` traffic, comparable to the results of the `rs-ip` ruleset and significantly lower than the results of the `rs-lpm` ruleset.

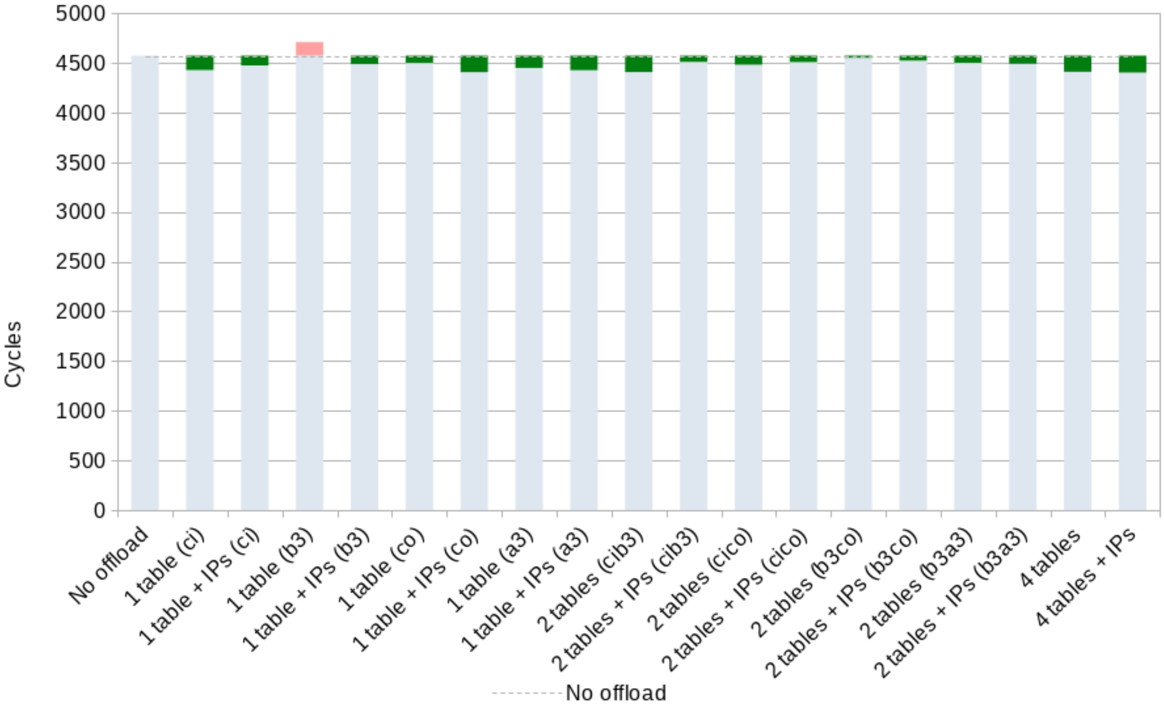
Listing 5.16: Ruleset `rs-mix` excerpt. All IP addresses are randomly generated.

```
...
count src 182.34.201.184
count dst 156.54.137.236
count src table gl2_country HR,US,JP,CN,SR
count dst 207.244.118.207
count src 106.75.133.10
...
```

Although the results in Figure 5.20 and Figure 5.21 show that metadata improves throughput in each case for `random` (up to 8.3% fewer cycles) and `specific` traffic (up to 3.7% fewer cycles), this improvement is still not large enough to compensate for the overall low throughput of this ruleset. Furthermore, IP address offload improves throughput for some table combinations, but worsens it for others.

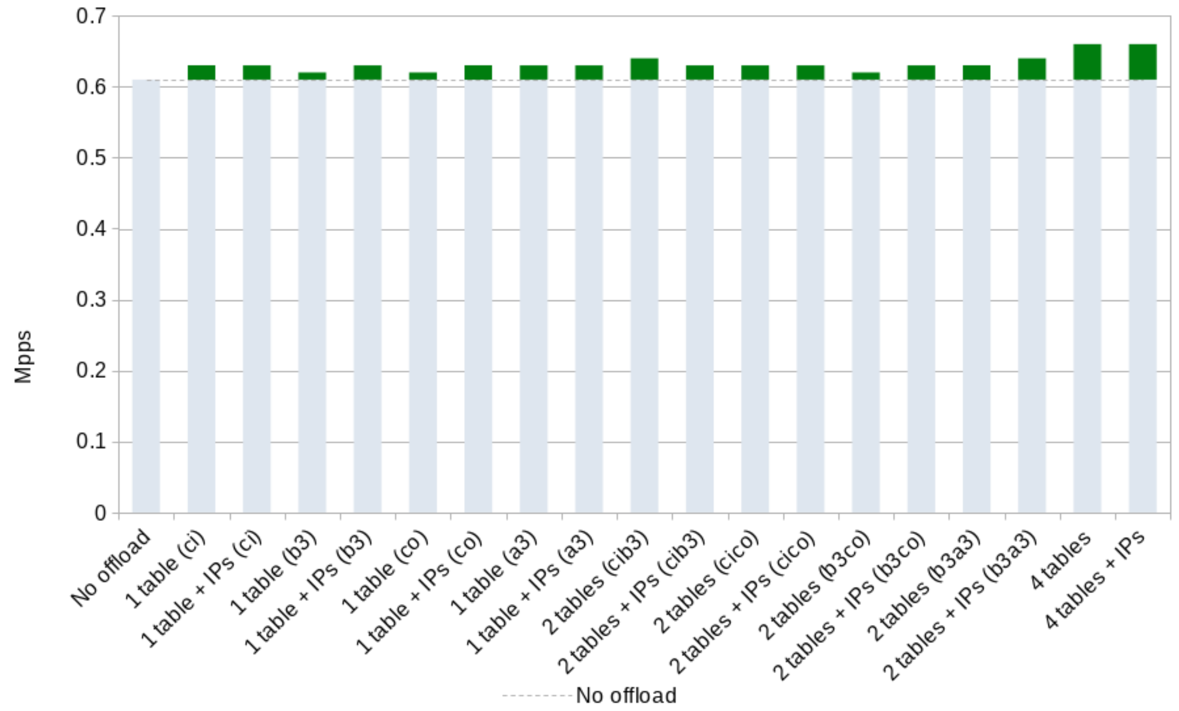


(a) Random traffic throughput

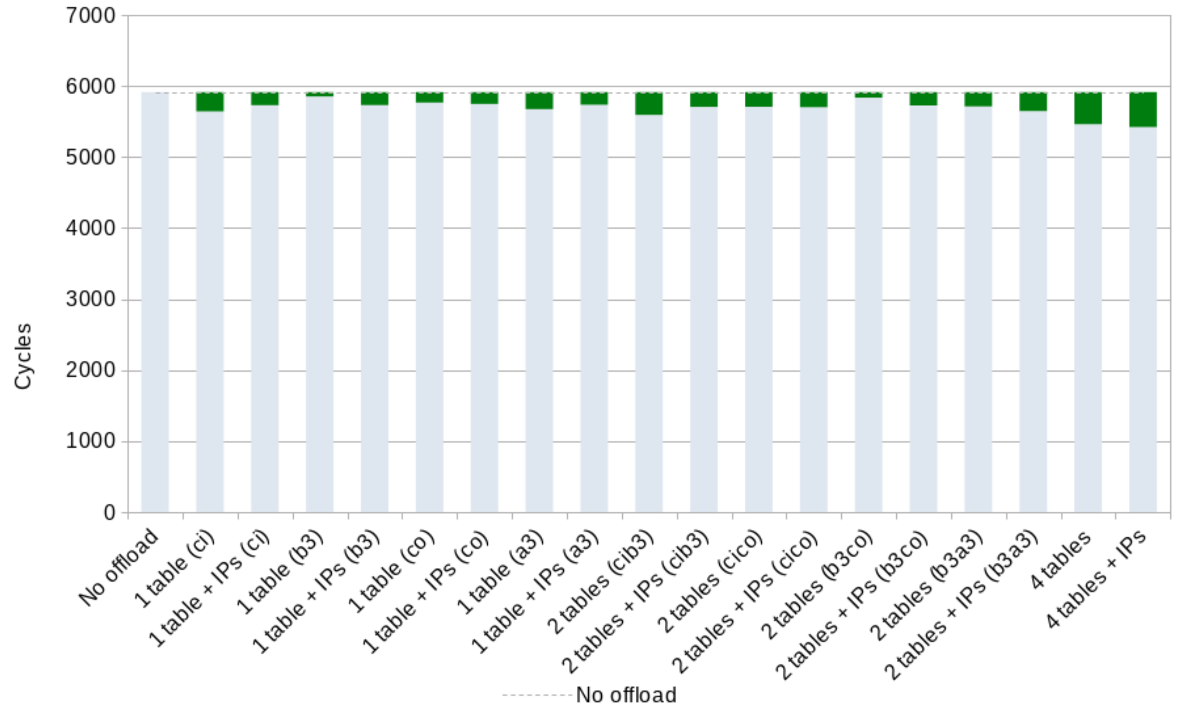


(b) Random traffic cycle count

Figure 5.20: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-mix` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.21: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-mix` for `specific` traffic.

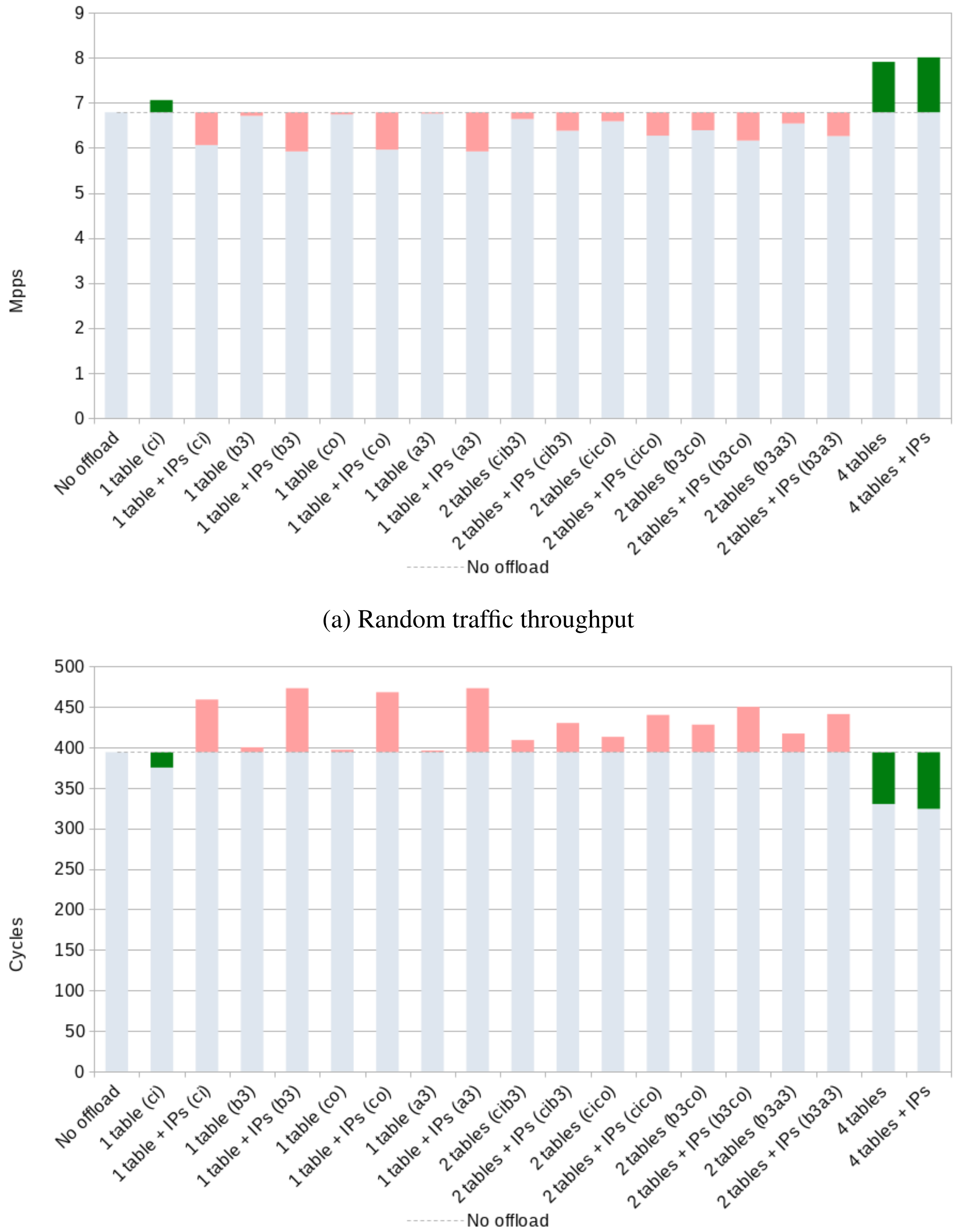
Ruleset `rs-lpmr`

This ruleset, shown in Listing 5.17, was used similarly to `rs-lpm` to test the scenario described in Section 4.1. The `rs-lpmr` ruleset includes *terminating* rules (ACCEPT and DENY) in addition to *counting* rules, so it is not comparable to other rulesets, but it shows results for the more realistic case of DDoS protection (inspired by the rulesets from Section 4.1), which none of the other tests do. Listing 5.17 also shows how variables can be defined and used. Variables are another feature of the software filter that can help organize rulesets and have better control over them.

Listing 5.17: Ruleset `rs-lpmr`.

```
include gl2_city.cfg    # 3,075,452 prefixes
include blocklist_3.cfg # 1,113,394 prefixes
include gl2_asn.cfg     # 430,976 prefixes
include gl2_country.cfg # 338,012 prefixes
include allowlist_3.cfg # 264,331 prefixes
include public.cfg      # 4 prefixes

define SUSP_CO CN,US,HK,GB
define SUSP_CI 1609776,1735158,9865869,6930379
define SUSP_ASN 133948,45528,4837,139007
define BALKAN_ALLOW HR,BA,SI,RS,ME

accept src table allowlist_3 ADMIN          # rule #1
deny not dst table public any               # rule #2
deny src table blocklist_3 BLOCK            # rule #3
count src table gl2_country $SUSP_CO        # rule #4
count src table gl2_city $SUSP_CI           # rule #5
count src table gl2_asn $SUSP_ASN           # rule #6
count src table blocklist_3 SUSP            # rule #7
accept src table allowlist_3 any            # rule #8
accept src table gl2_country $BALKAN_ALLOW  # rule #9
```

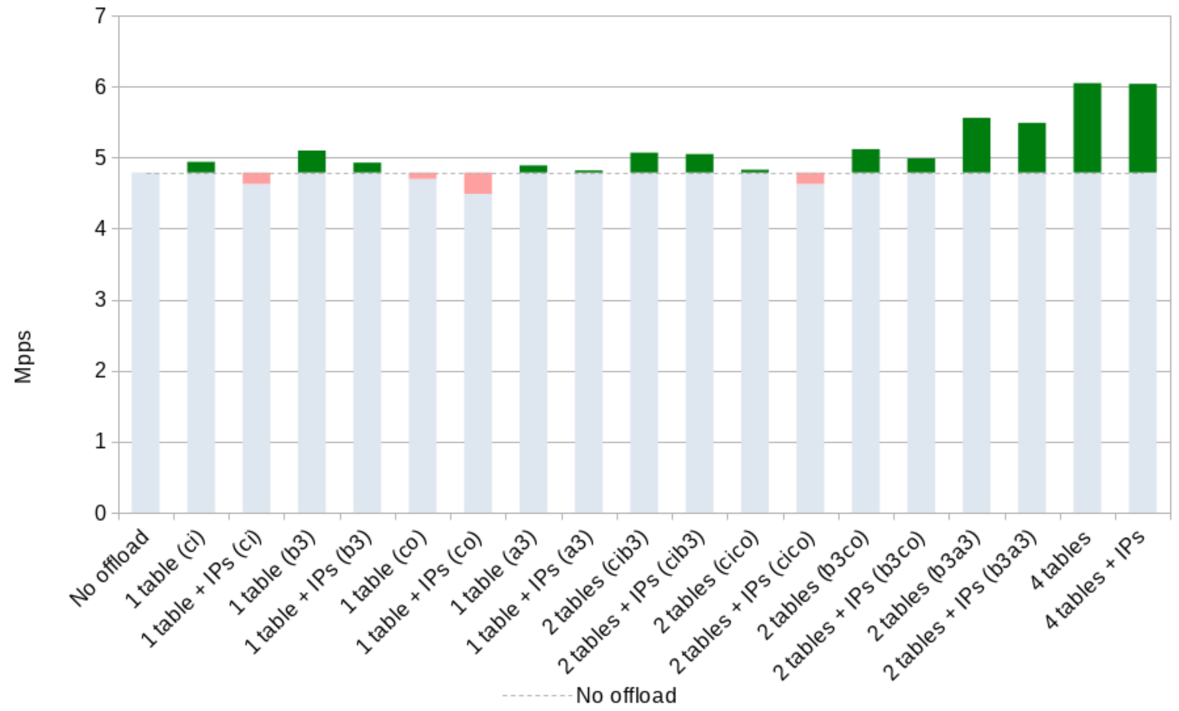
The results of this rulesset shown in Figure 5.22 and Figure 5.23 are different from the previous ones, especially with `random` traffic. It shows that the performance with offloading for this type of traffic does not change significantly in most cases and even deteriorates when IP addresses are included in the metadata. The most significant change is observed when four LPM tables are used for offloading (16.2% reduction in the number of cycles without IP address offloading and 17.8% with it). Another improvement was measured when only one LPM table (`gl2_city` — `ci`) was offloaded without using IP addresses in the metadata (4.8% reduction in the number of cycles). Due to the random distribution of the source IP addresses, the `gl2_city` table (the largest of tables used) has the highest probability of matching. In this case the size of the metadata does not significantly affect the software filter so offloading improves filtering performance.



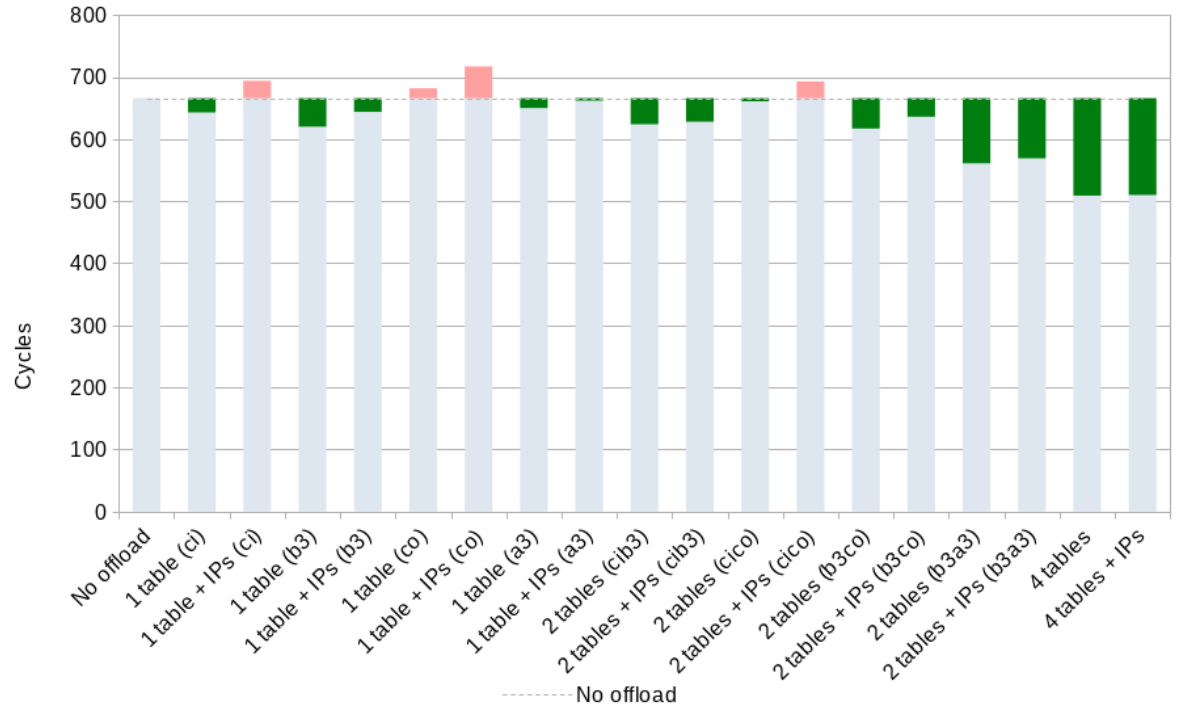
(a) Random traffic throughput

(b) Random traffic cycle count

Figure 5.22: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmr` for `random` traffic.



(a) Specific traffic throughput



(b) Specific traffic cycle count

Figure 5.23: Average filtering throughput and average cycle count of non-offloaded and offloaded ruleset `rs-lpmr` for `specific` traffic.

This assumption can be based on the results for `specific` traffic: since most traffic is generated from the blocklists and allowlists, the best improvement is for offloading the blocklist table (`blocklist_3` — `b3`) alone (6.9% reduction in cycles) and its combination with other tables (`gl2_country` — `b3co` with 7.4% and `allowlist_3` — `b3a3` with 15.8% reduction in cycles). Again, the greatest improvement is seen when using four tables (23.6% reduction in cycles without additional IP address offload and 23.4% with additional IP address offload).

5.2.1 Evaluation of the results

The results of the tests with the simulated hardware show that the inclusion of metadata can speed up the operation of the software filter for different types of traffic and for carefully selected types of metadata. The speedup is possible even though the overall packet size increases with the addition of metadata.

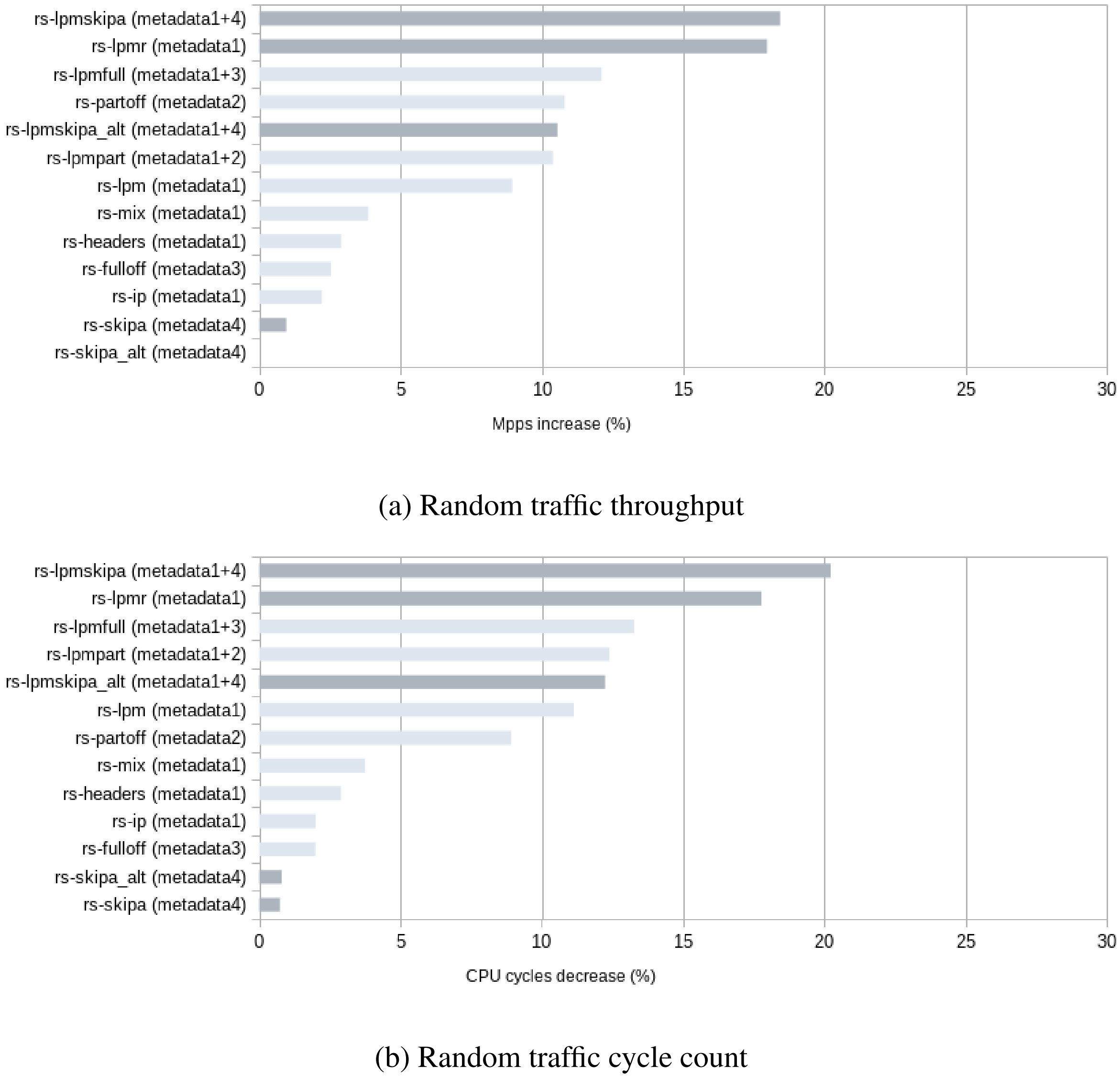
In most cases, moving rules (or parts of rules) from software to hardware results in a noticeable decrease in CPU load in software and an increase in overall throughput, and the more rules are offloaded to hardware, the better. However, the amount of metadata transferred between hardware and software can also affect filter performance, so the metadata type and its size must be considered before offloading rules. The constraints of the implementation used in this thesis do not allow a large number of rules to be offloaded, so it is not realistic to expect that large rulesets will be mostly filtered by hardware.

The baseline results of the rulesets with a "large" number of rules, shown in Figures 5.2 and 5.3, demonstrate the disadvantage of such rulesets in software filtering. As a reminder, 1000 IP addresses in a filter is often not enough when dealing with large volumetric DDoS attacks. Most of the CPU time is spent going through the rules and comparing them to the content of the packet, which is very time-consuming for software. The effectiveness of this approach decreases the larger the ruleset is. The throughput itself is very low, about 0.6–1.4 Mpps for each type of traffic. Offloading the metadata to the hardware provides some improvement, but not enough to noticeably increase throughput. However, this type of filtering needed to be tested to demonstrate its inefficiency and to measure the improvement when using hardware offloading.

Hybrid DDoS mitigation systems by other authors performed similar tests, assuming that the rulesets consist of a large number of rules belonging to $O**$ categories. In those cases, the rules are *fully Offloaded* to the hardware and the overall improvement is proportional to the number of rules offloaded to the hardware. Their conclusion fits to the results of this thesis: the more rules are offloaded to the hardware, the less work the software has to do and the higher the throughput. The hardware constraints limit this type of offloading to a number of rules that is not sufficient against volumetric DDoS attacks.

Although some improvements are more significant when offloading rulesets with a larger number of rules, it does not make sense to use these rulesets just because the performance improvement is higher. The highest baseline throughput (with the largest number of IP prefixes in a ruleset) is achieved when filtering with LPM lookups. It is obvious that this type of software filtering is suitable against volumetric DDoS attacks, and in most tests it was further improved with hybrid offloading. Even though some of the improvements do not seem to be very large, any improvement is important at such high speeds. Also, these types of rulesets make the firewall easier to manage.

Figure 5.24 and Figure 5.25 show the comparison of all tests as a maximum relative improvement over non-offloaded filtering for `random` and `specific` traffic. The figures include both types of rulesets: without `terminating` rules (light) and with `terminating` rules (dark).

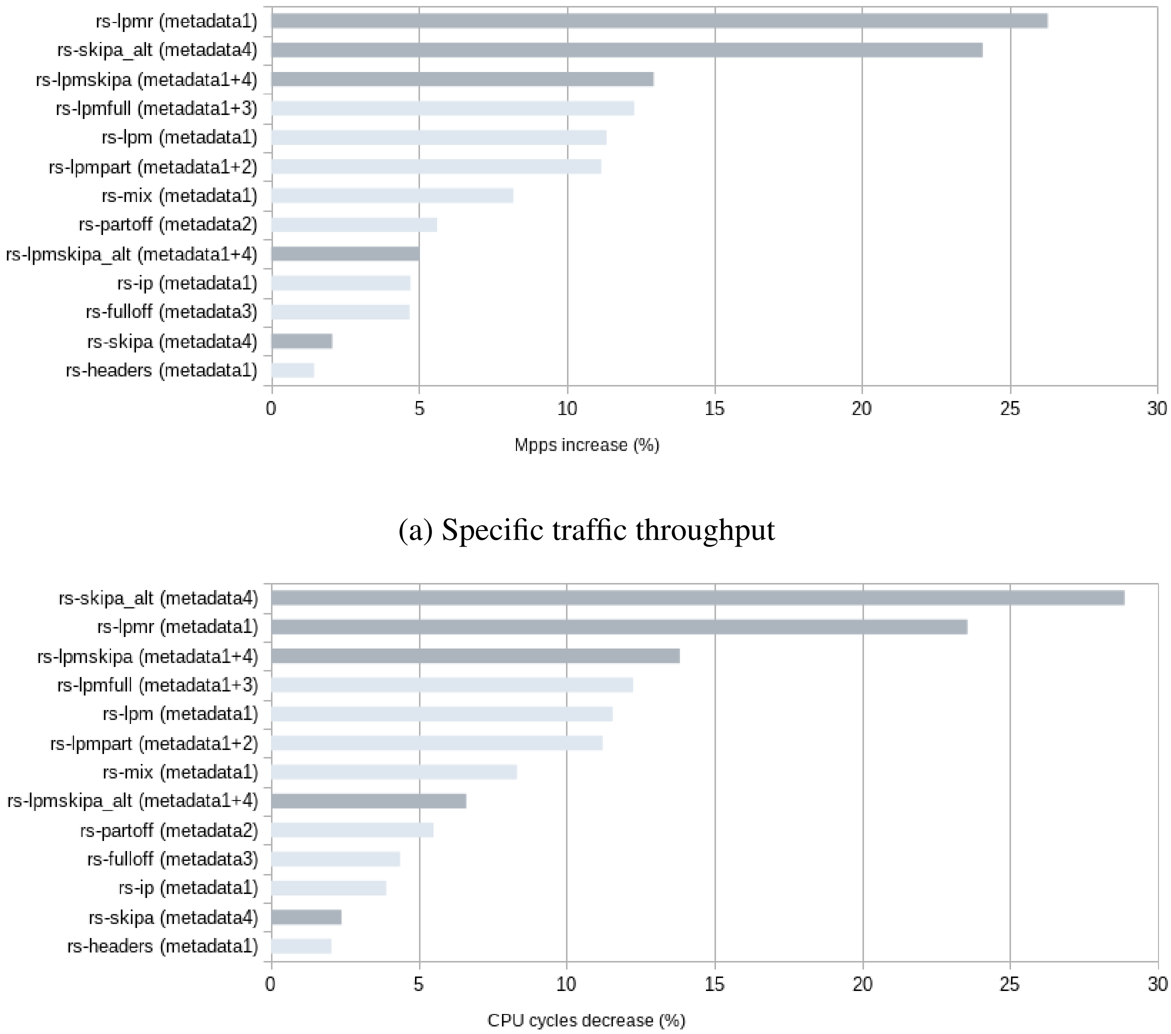


(a) Random traffic throughput

(b) Random traffic cycle count

Figure 5.24: Improvements in average filtering throughput and average cycle count of non-offloaded and offloaded rulesets for `random` traffic.

As expected, tests using rulesets with *terminating* rules show the greatest improvements. For `random` traffic (Figure 5.24), this is the `rs-lpmskipa` ruleset with 20.2% fewer cycles (18.4%



(a) Specific traffic throughput

(b) Specific traffic cycle count

Figure 5.25: Improvements in average filtering throughput and average cycle count of non-offloaded and offloaded rulesets for `specific` traffic.

increased throughput). For `specific` traffic (Figure 5.25), it is the `rs-skipa_alt` ruleset with 28.9% fewer cycles (24.1% increased throughput).

The second highest `random` traffic test uses the `rs-lpmr` ruleset, the “realistic” version of the `rs-lpm` ruleset that uses `terminating` rules. It achieved a 17.8% reduction in CPU cycles (18% increase in throughput). Other cases with the highest improvements for `random` traffic are those combining LPM with other metadata. They show a 12.2%–13.3% reduction in cycles (10.6%–12.1% increase in throughput). The `rs-lpm` ruleset (offloading four LPM tables with additional offloading of source IP addresses) achieved an 11.1% CPU cycle reduction (9% increase in throughput).

The rankings for `specific` traffic are similar to those for `random` traffic, with the second highest improvement achieved in tests using the `rs-lpmr` ruleset, with a 23.6% reduction in

CPU cycles (26.3% increase in throughput). As with `random` traffic, the other highest improvements are those combining LPM with other metadata, with a 13.8–11.2% reduction in cycles (11.2–13% increase in throughput). The `rs-lpm` ruleset shows an 11.6% reduction in cycles (11.4% increase in throughput).

Improvements for each individual test can scale depending on the capabilities of the hardware, i.e., performance can be further increased as more metadata is offloaded to the hardware. However, compared to other metadata types, offloading `metadata1` requires the fewest and least complex hardware updates to achieve a significant improvement in DDoS protection. For example, to increase the total number of available LPM tables for offloading to hardware, additional memory modules must be installed in the hardware. The use of the additional memory would not significantly change the internal FPGA logic and would not affect the overall performance of the system. For other metadata, more complex changes are required because parallelism cannot be used as effectively as in the case of `metadata1`. This would lead to delays and performance degradation, especially if a large number of rules are offloaded to hardware.

Improving the performance of the filter that uses hardware offloading means finding a balance between the size of the metadata and its usefulness. Responding to changes in the type and volume of traffic is also one of the most important matters to consider when offloading and even beforehand when creating the ruleset. Therefore, in cases where offloading has a negative impact on throughput, it can be bypassed and replaced with a better configuration.

It is worth repeating that all tests (including those performed with real hardware offloading) were performed on a system with a single CPU core at reduced frequency. Moreover, they were performed on a system corresponding to the model shown in Figure 3.2. The results of a hybrid system without the limitations of this model would certainly be even better.

5.3 Hybrid filter with metadata generated by hardware

The results shown in Section 5.2 are all based on tests using metadata pre-generated by the packet generator. To test the hardware part of the hybrid system, i.e., how the system works when the NetFPGA generates the metadata and attaches it to the packets, another set of tests was performed.

The test environment (testbed) for the hybrid tests, as shown in Figure 5.26, is similar to the one used when the hardware part was simulated by the software metadata generator. The traffic generator is connected to the NetFPGA SUME ingress interface, and the SUME egress interface is connected to the software filter ingress interface.

In all simulated tests, the packet generator typically sent packets in batches of about 256 and using four CPU threads (four NIC queues) to achieve sufficiently high speeds for certain tests (e.g., generating fully randomized packets and appending multiple metadata fields simultane-

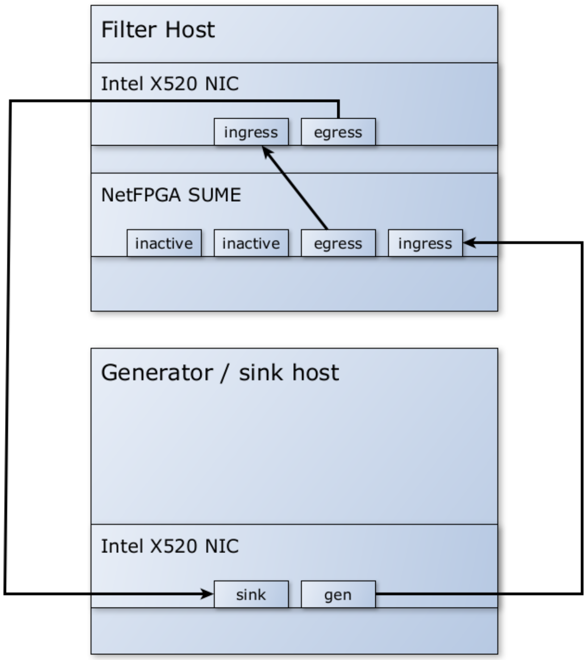


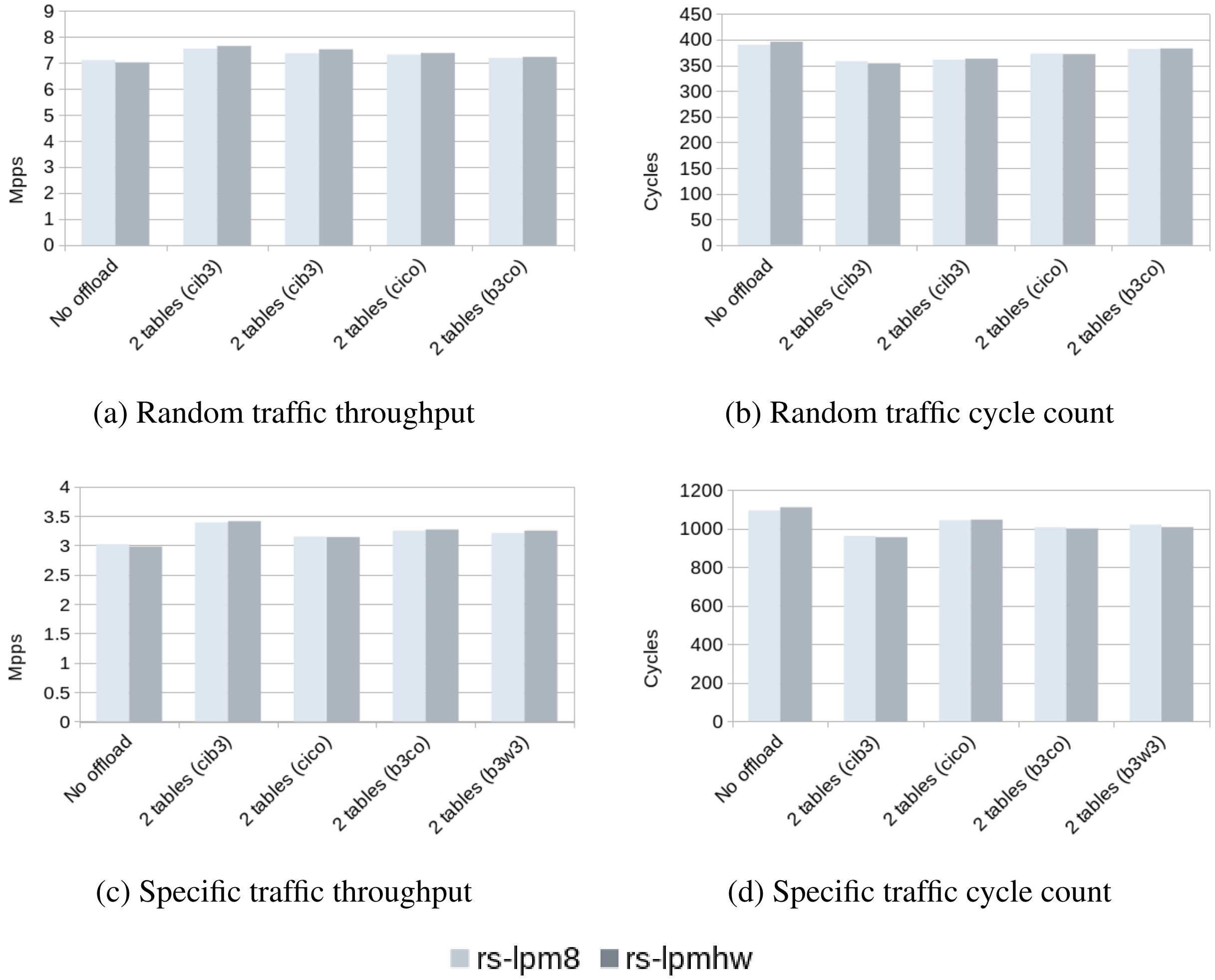
Figure 5.26: The testbed for testing the hybrid hardware / software system.

ously). Since the NetFPGA SUME version used in this thesis is not capable of processing traffic sent with these parameters, only one queue had to be used for these tests and the batch sizes were lowered. The tests with the same parameters had to be also performed for the simulated offload.

With these limitations, the tests for this hybrid system were performed in the same manner as the simulated tests, except that the packet generator did not have to create metadata and attach it to the packets. For this reason, it was expected that the results of all tests performed on the hybrid system would match the results of the simulated hardware. In all the tests that were performed with hybrid system, the average results of the hybrid system matched the average results of the simulation almost perfectly. From these results, it can be inferred that it is possible to achieve the same level of improvement over non-offloaded filtering with other types of metadata if specific offload capabilities were implemented on the hardware.

Figure 5.27 shows the results of one such test compared to the results of a test performed with pre-generated metadata. The test parameters for the `rs-lpm` ruleset were modified to account for the above limitations. As before, two tests were performed (baseline test and hardware offload test): multiple repetitions to obtain average values for throughput and CPU cycles. To distinguish them from the previous tests, the rulesets were named `rs-lpm8` and `rs-lpmhw`.

Filtering in hardware can be further improved by embedding `metadata4` offloading in the FPGA, by using a simple checker for a limited number of rules. For example, 16 port ranges for one of the previous rulesets can be stored in hardware memory, checked individually, and the results added to the metadata. This can be easily done by comparing the packet to a rule in



(a) Random traffic throughput

(b) Random traffic cycle count

(c) Specific traffic throughput

(d) Specific traffic cycle count

Figure 5.27: Average filtering throughput and average cycle count of non-offloaded and hardware offloaded ruleset `rs-lpm`.

each cycle and determining the value to include in the metadata. This is not suitable for large rulesets with a high number of rules that need to be checked, as this causes a much larger delay in the pipeline.

Additional hardware (in this case an FPGA) introduces some delay — and it can be even higher (for another clock cycle) if the last packet word and metadata exceed the 32-byte word limit of the AXI Stream, so another data word must be added. Latency is measured without high-precision methods and tools by simply sending and receiving the timestamped packet on the same computer and noting the difference between the send and receive times. It is important to note that both the source and destination are measured on the same computer and use the netmap framework to send and receive packets to bypass the overhead caused by the operating system. Also, the same CPU core is used so there is no additional time delay between sending and receiving. The delay caused by using the software filter is also shown (without any filtering, just by forwarding traffic from one interface to another). The results are shown in Figure 5.28 — the average value for forwarding through the NetFPGA (FPGA) is only $2\mu s$ higher than direct communication from interface to interface (IFC-IFC). The software filter increases the delay by about 400μs (SW filter).

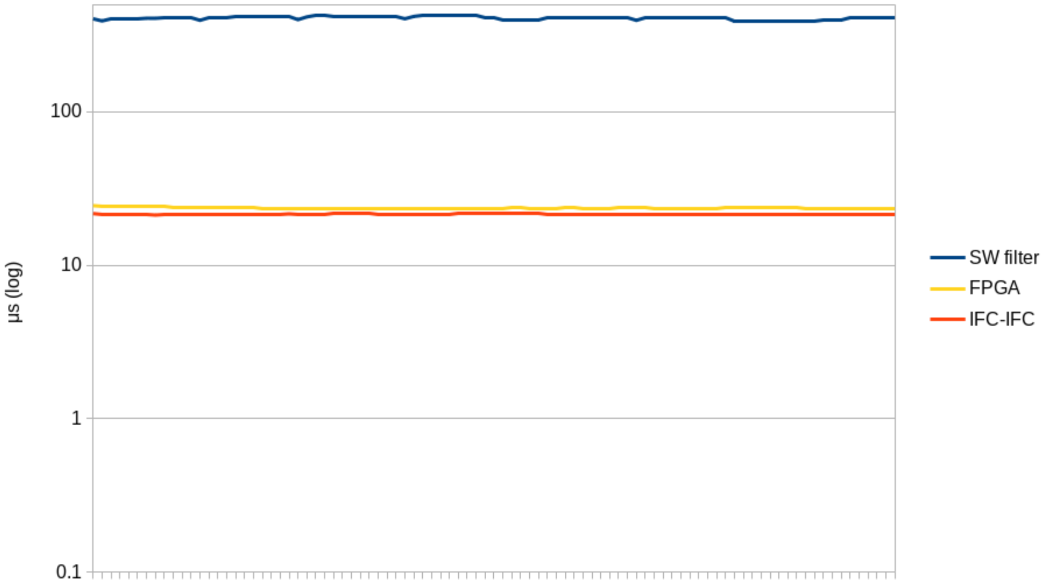


Figure 5.28: 10-value moving average of measured latencies for 3 different datapaths.

Chapter 6

Conclusion

Today's Internet is not only getting bigger, but also advancing technologically to the point where link speeds of 10 Gbit/s are becoming the norm for consumers, while 25, 40, 100 Gbit/s and higher are the de facto standard for ISPs and datacenters. With these speeds, devices need sufficient capacity to process the traffic that passes through them.

DDoS attacks are among the most common attacks on the Internet, and without proper defense mechanisms, volumetric DDoS attacks are very difficult and sometimes impossible to combat. This thesis demonstrates a novel approach to hybrid hardware / software filtering used against volumetric DDoS attacks. A model of a datapath for high-throughput network packet filtering is presented. It uses LPM for fast software filtering of IPv4 traffic and complements it with offloading parts of the algorithm to FPGA in order to achieve even higher performance.

The model of a filter was implemented by designing a configurable hardware datapath using FPGA technology and combining it with the existing software filter operating on commodity, off-the-shelf hardware. The hardware component of the model was implemented using NetFPGA SUME prototyping board. After processing each packet, FPGA generates and attaches specific metadata to it and forwards it to the software. The hardware datapath is generic and can be implemented on any FPGA with the necessary external components. The software filter from the previous research has been modified with additional functionality to receive metadata from the hardware and use it to filter packets with improved performance. Also, the NetFPGA SUME NIC driver for FreeBSD OS has been developed to improve the performance of hardware / software communication.

All possible types of rules used by the filter were categorized by their ability to be offloaded to hardware. Analysis of each category yielded appropriate metadata that the hardware could generate and forward to the software. This categorization served as the basis for a distributor model. The distributor is a part of the system that balances the distribution of rules to either hardware or software to achieve optimal overall throughput.

In order to evaluate different stages of filter development, an empirical method for testing this type of hybrid hardware / software system is developed. The method allows avoiding complex hardware design by simulating the hardware behavior without using real FPGA hardware. To do this, the existing packet generator was modified to automatically generate and attach metadata and send it to the software filter along with the packets. In this way, no FPGA modifications had to be made and all tests could be performed on a test environment consisting only of two off-the-shelf PCs.

One of the main contributions of this thesis is the custom reconfigurable datapath pipeline designed in NetFPGA SUME. A series of tests was performed on the hybrid system using the developed datapath, which confirmed the results obtained by the described simulations.

Various scenarios were empirically tested to investigate the best types of offloads and combinations of metadata to optimize DDoS protection. The tests consisted of scenarios with different rulesets (for different metadata types) while changing two parameters (traffic type and metadata size). This demonstrated the impact of rules belonging to different categories on the performance of the filter.

The implemented model shows performance improvements in tests with random traffic and traffic created specifically to simulate a DDoS attack. It is demonstrated that offloading different types of rules to hardware yields different performance gains, up to 30% fewer CPU cycles for certain offloads and types of rules. The benefits of using rules that leverage LPM in packet filtering are higher throughput and simpler, easier-to-manage rulesets. For this reason, these rulesets are best suited for DDoS protection, and they can be further increased with hardware offloading.

The implementation described in this thesis works primarily as a high-performance packet classifier / filter. As an additional feature, it provides counters for matching rules that can be used for statistical purposes and as input to existing (or new) external DDoS detection tools. These tools would complement the presented filter implementation and form an effective DDoS defense system that could filter out most of the malicious traffic arriving on a network. This would provide adequate protection against volumetric DDoS attacks before packets arrive to individual hosts, where they could be filtered more precisely, with slower, but stateful firewalls.

The hardware component from this thesis was implemented using the existing prototyping FPGA board which has constraints that affect the system performance. For this reason, only one of the possible models of a hybrid hardware / software filtering system is evaluated. The next research step would be to implement the proposed model using a hardware component without such limitations. NetFPGA PLUS, a recently released NetFPGA board capable of 100 Gbit/s networking, would serve as the hardware component for a potentially enhanced datapath. The proposed system has the potential for healthy scalability and a 100 Gbit/s datapath could provide a great testing environment. Since the hybrid system would use different hardware, scalability could lead to changes in implementation. The software filter is designed to be scalable, and previous research has shown that it works well in high-throughput networks. But at even higher speeds, the software could run into problems, so the overall system would benefit even more from the additional hardware offloading.

Since there are basically infinite combinations of rulesets, metadata offloading, different types of traffic, and ways for hardware and software to work together to filter traffic, more experiments can be conducted to evaluate them. In addition, there are various LPM algorithms, each with its own advantages and disadvantages and with different metadata for offloading. Experiments with other LPM algorithms could provide different performance improvements for certain types of DDoS traffic and potentially provide better insight into combating DDoS attacks.

Prototype evaluations used synthetic traffic specifically designed to resemble volumetric DDoS attacks. Although such traffic could theoretically represent a real DDoS attack, DDoS traffic from existing real network traces would paint a different picture when interpreting the results. These results would be more accurate and the resulting distributor model might be different from the one presented in this thesis. Testing the prototype by inserting it into an existing network would be an even better evaluation of the prototype as real traffic would be used, providing a basis for future work and a strong argument for continuing the research.

Bibliography

[1]Salopek, D., Vasi ć, V., Mikuc, M., “Security research and learning environment based on scalable network emulation.”, Tehnicki vjesnik/Technical Gazette, Vol. 24, 2017.

[2]“Exponential growth in ddos attack volumes”, https://cloud.google.com/blog/products/identity-security/identifying-and-protecting-against-the-largest-ddos-attacks, [Online; accessed 09-September-2021]. 2020.

[3]“Cisco annual internet report (2018–2023) white paper”, https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html, [Online; accessed 09-September-2021]. 2020.

[4]Mirkovic, J., Reiher, P., “A taxonomy of ddos attack and ddos defense mechanisms”, ACM SIGCOMM Computer Communication Review, Vol. 34, No. 2, 2004, pages 39–53.

[5]“Google ipv6 adoption statistics”, https://www.google.com/intl/en/ipv6/statistics.html#tab=ipv6-adoption, [Online; accessed 06-October-2021]. 2021.

[6]“Ams-ix traffic type statistics”, https://stats.ams-ix.net/sflow/ether_type.html, [Online; accessed 06-October-2021]. 2021.

[7]“Akamai ipv4 and ipv6 trends and statistics”, https://www.akamai.com/visualizations/dns-trends-and-traffic, [Online; accessed 06-October-2021]. 2021.

[8]Schutijser, C., “Comparing ddos mitigation techniques”, in 24th Twente Student Conference on IT, 2016, page 105.

[9]Kannan, K., Banerjee, S., “Compact tcam: Flow entry compaction in tcam for power aware sdn”, in International conference on distributed computing and networking. Springer, 2013, pages 439–444.

[10]Lakshminarayanan, K., Rangarajan, A., Venkatachary, S., “Algorithms for advanced packet classification with ternary cams”, ACM SIGCOMM Computer Communication Review, Vol. 35, No. 4, 2005, pages 193–204.

[11]Qu, Y. R., Prasanna, V. K., “Scalable and dynamically updatable lookup engine for decision-trees on fpga”, in 2014 IEEE High Performance Extreme Computing Conference (HPEC). IEEE, 2014, pages 1–6.

[12]Le, H., Prasanna, V. K., “Scalable high throughput and power efficient ip-lookup on fpga”, in 2009 17th IEEE Symposium on Field Programmable Custom Computing Machines. IEEE, 2009, pages 167–174.

[13]Yu, W., Sivakumar, S., Pao, D., “Pseudo-tcam: Sram-based architecture for packet classification in one memory access”, IEEE Networking Letters, 2019.

[14]Netfilter, L., “Iptables - linux man page”, https://linux.die.net/man/8/iptables, [Online; accessed 11-October-2021].

[15]Šimon, M., Huraj, L., Čerňanský, M., “Performance evaluations of iptables firewall solutions under ddos attacks”, Journal of Applied Mathematics, Statistics and Informatics, Vol. 11, No. 2, 2015, pages 35–45.

[16]Salopek, D., Vasi ć, V., Zec, M., Mikuc, M., Vašarević, M., Končar, V., “A network testbed for commercial telecommunications product testing”, in Software, Telecommunications and Computer Networks (SoftCOM), 2014 22nd International Conference on. IEEE, 2014, pages 372–377.

[17]Netfilter, L., “Ipset - linux man page”, https://linux.die.net/man/8/ipset, [Online; accessed 11-October-2021].

[18]Rizzo, L., “Netmap: a novel framework for fast packet i/o”, in 21st USENIX Security Symposium (USENIX Security 12), 2012, pages 101–112.

[19]Intel, D., “Intel data plane development kit”, http://dpdk.org/, [Online; accessed 09-September-2021]. 2014.

[20]Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M. V., “Creating complex network services with ebpf: Experience and lessons learned”, in 2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR). IEEE, 2018, pages 1–8.

[21]Kicinski, J., Viljoen, N., “ebpf hardware offload to smartnics: cls bpf and xdp”, Proceedings of netdev, Vol. 1, 2016.

[22]Bertin, G., “Xdp in practice: integrating xdp into our ddos mitigation pipeline”, in Technical Conference on Linux Networking, Netdev, Vol. 2, 2017.

[23]Deepak, A., Huang, R., Mehra, P., “ebpf/xdp based firewall and packet filtering”, in Linux Plumbers Conference, 2018.

[24]Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., Miller, D., “The express data path: Fast programmable packet processing in the operating system kernel”, in Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, 2018, pages 54–66.

[25]Hohlfeld, O., Krude, J., Reelfs, J. H., Rüth, J., Wehrle, K., “Demystifying the performance of xdp bpf”, in 2019 IEEE Conference on Network Softwarization (NetSoft). IEEE, 2019, pages 208–212.

[26]Kirdan, E., Raumer, D., Emmerich, P., Carle, G., “Building a traffic policer for ddos mitigation on top of commodity hardware”, in 2018 International Symposium on Networks, Computers and Communications (ISNCC). IEEE, 2018, pages 1–5.

[27]McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J., “Openflow: enabling innovation in campus networks”, ACM SIGCOMM computer communication review, Vol. 38, No. 2, 2008, pages 69–74.

[28]Molnár, L., Pongrácz, G., Enyedi, G., Kis, Z. L., Csikor, L., Juhász, F., K őrösi, A., Rétvári, G., “Dataplane specialization for high-performance openflow software switching”, in Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pages 539–552.

[29]Mauricio, L. A., Rubinstein, M. G., Duarte, O. C., “Proposing and evaluating the performance of a firewall implemented as a virtualized network function”, in 2016 7th International Conference on the Network of the Future (NOF). IEEE, 2016, pages 1–3.

[30]Peng, G., “Cdn: Content distribution network”, arXiv preprint cs/0411069, 2004.

[31]Imthiyas, M., Wani, S., Abdulghafor, R. A. A., Ibrahim, A. A., Mohammad, A. H., “Ddos mitigation: A review of content delivery network and its ddos defence techniques”, International Journal on Perceptive and Cognitive Computing, Vol. 6, No. 2, 2020, pages 67–76.

[32]Chen, M.-S., Liao, M.-Y., Tsai, P.-W., Luo, M.-Y., Yang, C.-S., Yeh, C. E., “Using netfpga to offload linux netfilter firewall”, in 2nd North American NetFPGA Developers Workshop. Citeseer, 2010.

[33]Li, B., Tan, K., Luo, L., Peng, Y., Luo, R., Xu, N., Xiong, Y., Cheng, P., Chen, E., “Clicknp: Highly flexible and high performance network processing with reconfigurable hardware”, in Proceedings of the 2016 ACM SIGCOMM Conference, 2016, pages 1–14.

[34]Fiessler, A., Hager, S., Scheuermann, B., Moore, A. W., “Hypafilter: A versatile hybrid fpga packet filter”, in Proceedings of the 2016 Symposium on Architectures for Networking and Communications Systems. ACM, 2016, pages 25–36.

[35]Fiessler, A., Lorenz, C., Hager, S., Scheuermann, B., Moore, A. W., “Hypafilter+: enhanced hybrid packet filtering using hardware assisted classification and header space analysis”, IEEE/ACM Transactions on Networking, Vol. 25, No. 6, 2017, pages 3655–3669.

[36]Go, Y., Jamshed, M. A., Moon, Y., Hwang, C., Park, K., “Apunet: Revitalizing GPU as packet processing accelerator”, in 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17), 2017, pages 83–96.

[37]Kalia, A., Zhou, D., Kaminsky, M., Andersen, D. G., “Raising the bar for using gpus in software packet processing”, in 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15), 2015, pages 409–423.

[38]Han, S., Jang, K., Park, K., Moon, S., “Packetshader: a gpu-accelerated software router”, ACM SIGCOMM Computer Communication Review, Vol. 41, No. 4, 2011, pages 195–206.

[39]Sun, W., Ricci, R., “Fast and flexible: parallel packet processing with gpus and click”, in Architectures for Networking and Communications Systems. IEEE, 2013, pages 25–35.

[40]Vasiliadis, G., Koromilas, L., Polychronakis, M., Ioannidis, S., “Gaspp: A gpu-accelerated stateful packet processing framework”, in 2014 USENIX Annual Technical Conference (USENIX ATC 14), 2014, pages 321–332.

[41]Miano, S., Doriguzzi-Corin, R., Risso, F., Siracusa, D., Sommese, R., “Introducing smartnics in server-based data plane processing: The ddos mitigation use case”, IEEE Access, Vol. 7, 2019, pages 107 161–107 170.

[42]Kaufmann, A., Peter, S., Sharma, N. K., Anderson, T., Krishnamurthy, A., “High performance packet processing with flexnic”, in Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems, 2016, pages 67–81.

[43]Cerovi ć, D., Del Piccolo, V., Amamou, A., Haddadou, K., Pujolle, G., “Fast packet processing: A survey”, IEEE Communications Surveys & Tutorials, Vol. 20, No. 4, 2018, pages 3645–3676.

[44]Hager, S., Winkler, F., Scheuermann, B., Reinhardt, K., "Mpfc: Massively parallel firewall circuits", in 39th Annual IEEE Conference on Local Computer Networks. IEEE, 2014, pages 305–313.

[45]"Enhancing networks with smartnics", https://www.mellanox.com/files/doc-2020/sb-smart-nic.pdf, [Online; accessed 09-September-2021]. 2020.

[46]AL-Musawi, B. Q. M., "Mitigating dos/ddos attacks using iptables", International Journal of Engineering & Technology, Vol. 12, No. 3, 2012, pages 101–111.

[47]"Read dyn's statement on the 10/21/2016 dns ddos attack | dyn blog", https://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/, [Online; accessed 18-January-2020]. 2016.

[48]"How to not break the internet", https://www.kaspersky.com/blog/attack-on-dyn-explained/13325/, [Online; accessed 09-September-2021]. 2016.

[49]"Dyn (dyndns) ddos attack", https://www.red-button.net/blog/dyn-dyndns-ddos-attack/, [Online; accessed 09-September-2021]. 2016.

[50]"Massive cyber attack 'sophisticated, highly distributed', involving millions of ip addresses", https://www.cnbc.com/2016/10/22/ddos-attack-sophisticated-highly-distributed-involved-millions-of-ip-addresses-dyn.html, [Online; accessed 09-September-2021]. 2016.

[51]Putnam, A., Caulfield, A. M., Chung, E. S., Chiou, D., Constantinides, K., Demme, J., Esmaeilzadeh, H., Fowers, J., Gopal, G. P., Gray, J. *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services", ACM SIGARCH Computer Architecture News, Vol. 42, No. 3, 2014, pages 13–24.

[52]Cholakoska, A., Efnusheva, D., Kalendar, M., "Hardware implementation of ip packet filtering in fpga", in Proceedings of the 7th International Conference on Applied Innovations in IT, 2019.

[53]Ajami, R., Dinh, A., "Embedded network firewall on fpga", in 2011 Eighth International Conference on Information Technology: New Generations. IEEE, 2011, pages 1041–1043.

[54]Sourdis, I., Dimopoulos, V., Pnevmatikatos, D., Vassiliadis, S., "Packet pre-filtering for network intrusion detection", in 2006 Symposium on Architecture For Networking And Communications Systems. IEEE, 2006, pages 183–192.

[55]"Snort home page", https://www.snort.org/, [Online; accessed 06-October-2021]. 2021.

[56]Pontarelli, S., Bianchi, G., Teofili, S., “Traffic-aware design of a high-speed fpga network intrusion detection system”, IEEE Transactions on Computers, Vol. 62, No. 11, 2013, pages 2322–2334.

[57]Attig, M., Brebner, G., “400 gb/s programmable packet parsing on a single fpga”, in 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems. IEEE, 2011, pages 12–23.

[58]Tokusashi, Y., Matsutani, H., Zilberman, N., “Lake: An energy efficient, low latency, accelerated key-value store”, arXiv preprint arXiv:1805.11344, 2018.

[59]Nishtala, R., Fugal, H., Grimm, S., Kwiatkowski, M., Lee, H., Li, H. C., McElroy, R., Paleczny, M., Peek, D., Saab, P. *et al.*, “Scaling memcache at facebook”, in 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13), 2013, pages 385–398.

[60]Taylor, D. E., “Survey and taxonomy of packet classification techniques”, ACM Computing Surveys (CSUR), Vol. 37, No. 3, 2005, pages 238–275.

[61]Vamanan, B., Voskuilen, G., Vijaykumar, T., “Efficuts: Optimizing packet classification for memory and throughput”, ACM SIGCOMM Computer Communication Review, Vol. 40, No. 4, 2010, pages 207–218.

[62]Li, W., Li, X., “Hybridcuts: A scheme combining decomposition and cutting for packet classification”, in 2013 IEEE 21st Annual Symposium on High-Performance Interconnects. IEEE, 2013, pages 41–48.

[63]Li, W., Li, X., Li, H., Xie, G., “Cutsplit: A decision-tree combining cutting and splitting for scalable packet classification”, in IEEE INFOCOM 2018-IEEE Conference on Computer Communications. IEEE, 2018, pages 2645–2653.

[64]Taylor, D. E., Turner, J. S., “Classbench: A packet classification benchmark”, IEEE/ACM transactions on networking, Vol. 15, No. 3, 2007, pages 499–511.

[65]Zec, M., Mikuc, M., “Pushing the envelope: Beyond two billion ip routing lookups per second on commodity cpus”, in 2017 25th International Conference on Software, Telecommunications and Computer Networks (SoftCOM). IEEE, 2017, pages 1–6.

[66]Asai, H., Ohara, Y., “Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup”, in ACM SIGCOMM Computer Communication Review, Vol. 45, No. 4. ACM, 2015, pages 57–70.

[67]Yang, T., Xie, G., Liu, A. X., Fu, Q., Li, Y., Li, X., Mathy, L., "Constant ip lookup with fib explosion", IEEE/ACM Transactions on Networking, Vol. 26, No. 4, 2018, pages 1821–1836.

[68]Huston, G., "Bgp routing table analysis reports", https://www.cidr-report.org/as2.0/, [Online; accessed 23-September-2021]. 2021.

[69]Zec, M., "Improving performance in software internet routers through compact lookup structures and efficient datapaths", PhD thesis, University of Zagreb. Faculty of Electrical Engineering and Computing, 2019.

[70]Salopek, D., Zec, M., Mikuc, M., Vasi ć, V., "Surgical ddos filtering with fast lpm", IEEE Access, 2021.

[71]Qu, Y. R., Prasanna, V. K., "High-performance and dynamically updatable packet classification engine on fpga", IEEE Transactions on Parallel and Distributed Systems, Vol. 27, No. 1, 2015, pages 197–209.

[72]Li, C., Li, T., Li, J., Li, D., Yang, H., Wang, B., "Memory optimization for bit-vector-based packet classification on fpga", Electronics, Vol. 8, No. 10, 2019, page 1159.

[73]Watson, G., McKeown, N., Casado, M., "Netfpga: A tool for network research and education", in 2nd workshop on Architectural Research using FPGA Platforms (WARFP), Vol. 3, 2006.

[74]Lockwood, J. W., McKeown, N., Watson, G., Gibb, G., Hartke, P., Naous, J., Raghuraman, R., Luo, J., "Netfpga–an open platform for gigabit-rate network switching and routing", in 2007 IEEE International Conference on Microelectronic Systems Education (MSE'07). IEEE, 2007, pages 160–161.

[75]Zilberman, N., Audzevich, Y., Kalogeridou, G., Manihatty-Bojan, N., Zhang, J., Moore, A., "Netfpga: Rapid prototyping of networking devices in open source", ACM SIGCOMM Computer Communication Review, Vol. 45, No. 4, 2015, pages 363–364.

[76]Zilberman, N., Audzevich, Y., Covington, G. A., Moore, A. W., "Netfpga sume: Toward research commodity 100gb/s", 2014.

[77]Lai, Y.-K., Huang, P.-Y., Lee, H.-P., Tsai, C.-L., Chang, C.-S., Nguyen, M. H., Lin, Y.-J., Liu, T.-L., Chen, J. H., "Real-time ddos attack detection using sketch-based entropy estimation on the netfpga sume platform", in 2020 Asia-Pacific Signal and Information Processing Association Annual Summit and Conference (APSIPA ASC). IEEE, 2020, pages 1566–1570.

[78]Gondaliya, H., Sankaran, G. C., Sivalingam, K. M., “Comparative evaluation of ip address anti-spoofing mechanisms using a p4/netfpga-based switch”, in Proceedings of the 3rd P4 Workshop in Europe, 2020, pages 1–6.

[79]Nurmi, J., Zolfaghari, D. S. H., “Prototyping 40g ethernet communication on fpga”, 2021.

[80]Rodrigues, P., Saquetti, M., Bueno, G., Cordeiro, W., Azambuja, J., “Virtualization of programmable forwarding planes with p4vbox”, Journal of Integrated Circuits and Systems, Vol. 16, No. 2, 2021, pages 1–8.

[81]“Netfpga-sume-live github repository”, https://github.com/NetFPGA/NetFPGA-SUME-live, [Online; accessed 06-October-2021]. 2020.

[82]“Riffa github repository”, https://github.com/KastnerRG/riffa, [Online; accessed 06-October-2021]. 2016.

[83]FreeBSD, “Freebsd kernel developer’s manual”, https://www.freebsd.org/cgi/man.cgi?query=iflib&sektion=9&format=html, [Online; accessed 02-October-2021]. 2018.

[84]Zazo, J. F., Lopez-Buedo, S., Audzevich, Y., Moore, A. W., “A pcie dma engine to support the virtualization of 40 gbps fpga-accelerated network appliances”, in 2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig). IEEE, 2015, pages 1–6.

[85]Forencich, A., Snoeren, A. C., Porter, G., Papen, G., “Corundum: An open-source 100-gbps nic”, in 2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). IEEE, 2020, pages 38–46.

[86]Ujjan, R. M. A., Pervez, Z., Dahal, K., Bashir, A. K., Mumtaz, R., González, J., “Towards sflow and adaptive polling sampling for deep learning based ddos detection in sdn”, Future Generation Computer Systems, Vol. 111, 2020, pages 763–779.

[87]Erhan, D., Anarim, E., “Hybrid ddos detection framework using matching pursuit algorithm”, IEEE Access, Vol. 8, 2020, pages 118 912–118 923.

[88]Cviti ć, I., Peraković, D., Periša, M., Botica, M., “Novel approach for detection of iot generated ddos traffic”, Wireless Networks, Vol. 27, No. 3, 2021, pages 1573–1586.

[89]Doshi, R., Apthorpe, N., Feamster, N., “Machine learning ddos detection for consumer internet of things devices”, in 2018 IEEE Security and Privacy Workshops (SPW). IEEE, 2018, pages 29–35.

[90]Gupta, P., Lin, S., McKeown, N., “Routing lookups in hardware at memory access speeds”, in Proceedings. IEEE INFOCOM’98, the Conference on Computer Communications. Seventeenth Annual Joint Conference of the IEEE Computer and Communications Societies. Gateway to the 21st Century (Cat. No. 98), Vol. 3. IEEE, 1998, pages 1240–1247.

[91]Intel, “Intel 64 and ia-32 architectures software developer’s manual volume 2b: Instruction set reference, m-z”, http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-vol-2b-manual.pdf, [Online; accessed 06-October-2021]. 2014.

Curriculum Vitae

Denis Salopek was born in 1990 in Ogulin. He obtained the B.Sc. and M.Sc. degrees in Computer Science from the University of Zagreb, Faculty of Electrical Computing and Engineering, Zagreb, Croatia, in 2011 and 2013, respectively. He is currently pursuing his Ph.D. degree of Electrical Engineering and Computing at the same university under the supervision of Associate Professor Miljenko Mikuc.

From 2013 to 2016, he worked as a research assistant in the Department of Telecommunications at the Faculty of Electrical Computing and Engineering on a project in the field of information and communication technology in collaboration with Ericsson Nikola Tesla d.d., "Customized IMUNES for Ericsson (E-IMUNES)". He was one of the developers of the IMUNES network emulator / simulator and extended the standard functionalities of the tool for the purposes of the project.

Since then, he has been a teaching assistant at the same faculty, collaborating on four courses. Besides his duties as a teaching assistant, he also participates in the project "Smart human-centric services in interoperable and decentralised IoT environments" (IoT4us). He is currently the main developer of IMUNES, which is used worldwide as a learning tool for communication networks and their protocols. His research interests also include high-speed networking, FPGA, virtualization, and kernel programming. He has contributed to the FreeBSD kernel by porting the Linux version of the NetFPGA SUME NIC driver, and to the Linux kernel with his work on extending eBPF functionalities.

Denis Salopek has published four peer-reviewed papers in international journals and conferences.

List of publications

Journals

- Salopek D., Vasi ć V., Mikuc M. "Security Research and Learning Environment Based on Scalable Network Emulation", Tehnički vjesnik/Technical Gazette, September 2017.
- Salopek D., Zec M, Mikuc M., Vasi ć V. "Surgical DDoS Filtering with Fast LPM", IEEE Access, January 2022.

Conference proceedings

- Salopek, D., Vasi ć, V., Zec M., Mikuc, M., Vašarević, M., Končar, V. “A network testbed for commercial telecommunications product testing”, SoftCOM, Split, 2014.
- Vasi ć, V., Salopek D., Pripužić K, Vuković M., “Sustav aktivnog otkrivanja uljeza na sjedištu WWW.HR”, Carnet Users Conference CUC, Zagreb, 2015.

Životopis

Denis Salopek rođen je 1990. godine u Ogulinu. Na Fakuletu elektrotehnike i računarstva Sveučilišta u Zagrebu 2011. godine završio je preddiplomski studij Računarstvo te 2013. godine diplomski studij Informacijska i komunikacijska tehnologija. Trenutno je upisan na doktorski studij Fakulteta pod mentorstvom izv. prof. dr. sc. Miljenka Mikuca.

Od 2013. do 2016. godine radio je kao znanstveni suradnik na Zavodu za telekomunikacije Fakulteta elektrotehnike i računarstva na istraživačkom projektu u sklopu suradnje na području informacijskih i komunikacijskih tehnologija s kompanijom Ericsson Nikola Tesla d.d., "Prilagođen IMUNES za Ericsson (E-IMUNES)". Bio je jedan od programera IMUNES mrežnog emulatora / simulatora i proširivao standardne funkcionalnosti alata za potrebe projekta.

Od tada radi kao asistent na istom fakultetu, te u toj funkciji sudjeluje na četiri kolegija. Osim dužnosti u nastavi, sudjeluje u projektu "Pametne usluge usmjerene čovjeku u interoperabilnim i decentraliziranim okolinama Interneta stvari" (IoT4us). Trenutno je glavni razvojni programer alata za emuliranje mreže IMUNES, koji se u nekoliko kolegija na FER-u, ali i širom svijeta, koristi kao alat za učenje komunikacijskih mreža i njihovih protokola. Osim navedenog, fokus njegovog istraživanja je na brzim mrežama, FPGA tehnologiji, virtualizaciji te programiranju u jezgri operacijskih sustava. Pridonio je jezgri operacijskog sustava FreeBSD prilagodbom Linux verzije upravljača za NetFPGA SUME NIC i jezgri Linux operacijskog sustava svojim radom na proširenju eBPF funkcionalnosti.

Objavio je četiri recenzirana rada u međunarodnim časopisima i konferencijama.