VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA PRO RYCHLÉ ZPRACOVÁNÍ SÍŤOVÝCH DAT

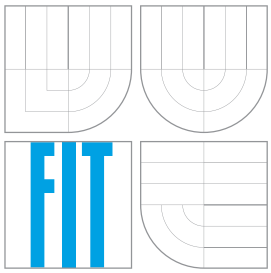
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ VOKRÁČKO

BRNO 2015

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KNIHOVNA PRO RYCHLÉ ZPRACOVÁNÍ SÍŤOVÝCH DAT

LIBRARY FOR FAST NETWORK TRAFFIC PROCESSING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — LUKÁŠ VOKRÁČKO
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KOŘENEK, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

Tato práce se zabývá časově kritickými operacemi v oblasti počítačových sítích a zahrnuje návrh API pro knihovnu implementující tyto operace. Mezi zpracované operace patří vyhledání nejdelšího shodného prefixu pomocí algoritmů TreeBitmap a Binary search on prefix length, hledání řetězců algoritmem Aho-Corasick, hledání regulárních výrazů, analýza a extrakce hlaviček paketů a klasifikace paketů. V práci je zhodnocena dosažená rychlost implementace těchto operací na platformách Intel a ARM.

Abstract

Výtah (abstrakt) práce v anglickém jazyce.

Klíčová slova

počítačové sítě, hledání nejdelšího shodného prefixu, hledání řetězců, regulární výrazy, binární vyhledávání na prefixu, treebitmap, aho-corasick

Keywords

Klíčová slova v anglickém jazyce.

Citace

Lukáš Vokráčko: Knihovna pro rychlé zpracování síťových dat, bakalářská práce, Brno, FIT VUT v Brně, 2015

Knihovna pro rychlé zpracování síťových dat

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana ...

.....
Lukáš Vokráčko
3. května 2015

Poděkování

Zde je možné uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc.

Obsah

1 Úvod **2**

2 Teoretický rozbor **4**
2.1 Síťové modely 4
2.1.1 Fyzická vrstva / Vrstva síťového rozhraní 4
2.1.2 Linková vrstva 5
2.1.3 Síťová vrstva 5
2.1.4 Transportní vrstva 5
2.2 Časově kritické operace 5
2.2.1 Hledání nejdelšího společného prefixu - Longest prefix matching . . . 6
2.2.2 Hledání řetězců 9
2.2.3 Hledání regulárních výrazů 10
2.2.4 Analýza a extrakce hlaviček paketů - Header parsing 11

3 Návrh API knihovny **12**
3.1 Vyhledání nejdelšího shodného prefixu 12
3.2 Hledání řetězců 13
3.3 Regulární výrazy 14
3.4 Použití knihovny 15
3.5 Rošíření knihovny 15
3.5.1 Packer header extraction 15
3.5.2 Packer clasification 16

4 Výsledky **17**
4.1 Hledání nejdelšího shodného prefixu 17

5 Závěr **20**
5.1 Další rozšíření 20
5.1.1 Optimalizace 21
5.1.2 Stress-testing 21

Kapitola 1

Úvod

Žijeme v době kdy se internet stal nedílnou součástí každodenního života a s internetem už nepracují pouze klasické počítače, ale do popředí se také dostávají mobilní zařízení, které meziročně zaznamenávají více než 50% nárůst. Dalším druhem zařízení jež se začínají připojovat jsou vestavěné systémy patřící do trendu nazývaného internet věcí. S rychlostí jakou přibývají zařízení vyžadující přístup k internetu ale i s rozšiřováním internetu do zemí internetem nedotčených se neustále zvyšují požadavky na rychlost, se kterou data prochází počítačovými sítěmi a z toho vyplívající požadavky na rychlost zpracování síťového provozu a to zejména na zařízeních starajících se o řízení internetového provozu na páteřních linkách počítačových sítí. Mezi tyto zařízení lze zařadit směrovače (routery), které řídí datové toky mezi jednotlivými sítěmi, rozbočovače (switche) starající se o řízení toků dat uvnitř autonomních sítí a systémy pro detekci (IDS[1]) a prevenci (IPS[2]) síťových útoků, které analyzují obsah každého paketu, který sítí prochází. Páteřní spoje v době psaní této práce dosahují rychlostí v řádech gigabitů za sekundu a z toho vyplívají požadavky na rychlost zpracování síťových dat. Při těchto rychlostech však tradiční procesory stihnou vykonat pouze desítky instrukcí, které nestačí na provedení včech potřebných operací. Nicméné je důležité aby stejné rychlosti zpracování dosahovaly všechna zařízení na páteřních spojích, protože v počítačová sít je pouze tak rychlá, jak rychlá je její nejpomalejší část, tzv. úzké hrdlo. Z těchto vlastností vycházejí požadavky na stále efektivnější algoritmy zpracovávající časově kritické operace, jejichž rozbor je součástí této práce. Z těchto operací jsou vybrány a rozvedeny operace prohledávání vstupních dat na přítomnost definovaných slov pomocí vyhledávání řetězců a hledání regulárních výrazů pro hledání signatur útoků v počítačovém provozu, analýza a extrakce hlaviček paketů - operace používané napříč téměř všemi časově kritickými operacemi a klasifikace paketů, speciálně pak jednodimenzionální klasifikace podle cílové IP adresy, hledání nejdelšího shodného prefixu, jež je využíváno pro prohledávání směrovací tabulek směrovačů pro nalezení takové cesty počítačovou sítí, která bude nejhodnější.

Přínosem této knihovny je vlastní implementace zmíněných časově kritických operací, která bude využita výkumnou skupinou ANT na Fakultě Informačních technologií Vysokého učení technického v Brně pro ??????????? a také může sloužit jako vzor pro hardwarovou akceleraci procházení a vyhledávání ve zmíněných operací buď syntézou kódu do programovatelných hradlových polí FPGA[3] nebo pro návrh samostatnéhých integrovaných obvodů

[1] Intrusion detection system
[2] Intrusion prevention system
[3] Field Programmable Gate Array

ASIC[4]. Pro knihovnu je navržen a implementován obecný princip vláknového zpracování použitelný pro všechny zmíněné operace, který umožňuje řetězit požadavky na zpracování do vyrovnávacích pamětí a jednotlivé operace provádět plynule za sebou bez nutnosti zasahovat do datových struktur nebo toku řízení.

V kapitole 2 jsou popsány síťové modely a vrstvy těchto modelů, nad nimiž jsou operace této knihovny implementovány, dále jsou popsány časově kritické operace prováděné prvky v počítačových sítích. Kapitola 3 popisuje návrh veřejného rozhraní vytvořené knihovny pro operace zmíněné v kapitole 2, způsoby použití potéto knihovny a možnosti rozšíření o další časově kritické operace. V kapitole 4 jsou vizualizovány a diskutovány výsledky, jež se podařilo dosáhnout v implementaci této knihovny a to na dvou hlavních platformách, Intel a ARM. Kapitola 5 shrnuje dosažené výsledky a nastiňuje další možný vývoj této knihovny a to jak z hlediska rozšiřování repertoáru implementovaných operací tak i z pohledu dalších optimalizací a testování v různých scénářích při reálném používání knihovny.

Co zmínit - všchny prvky musí pracovat na stejných rychlostech - zvyšuje se rychlost -¿ nutná optimalizace, fpga - požadavky na monitorování a blokování - ipv4, ipv6, velikost adresového prostoru - pomalé procesory -¿ směr fpga, jak je na to knihovna přpravena - koncept multithreadingu - hlavní síťové prvky - páteřní sítě - pakety - půlstrana motivace proč je to důležité - přínos knihovny a operace - vlákna - plná fronta nebo timeout pro spuštění

[4] Application Specific Integrated Circuit

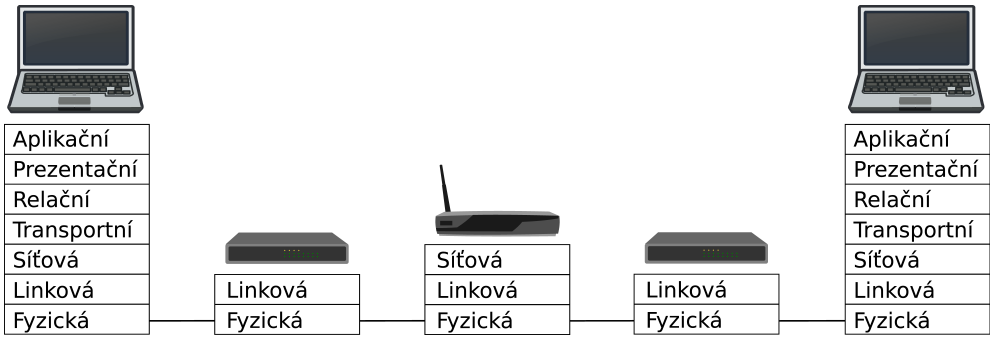
Kapitola 2

Teoretický rozbor

Co zmínit - složitost síťového provozu, proto vrstvy - něco o kritických operacích - proč jsou důležité - popsat další kapitoly

2.1 Síťové modely

Zpracování dat síťového provozu je rozděleno do několika úrovní. Tyto úrovně jsou popsány síťovými modely. Základním modelem je ISO/OSI, který slouží pro abstraktní rozdělení operací zpracování síťových dat a jeho použití je pouze pro akademické účely. V reálných počítačových sítích pak dominuje model TCP/IP, který má oproti ISO/OSI modelu menší počet vrstev. Modelu ISO/OSI je rozdělen na sedm vrstev. V pořadí od nejnižší úrovně to jsou vrstvy fyzická, linková, síťová, transportní, relační, prezentační a aplikační. Pro tuto práci jsou podstatné pouze první čtyři vrsty. Ty jsou detailněji popsány v kapitolách 2.1.1, 2.1.2, 2.1.3 a 2.1.4. Na obrázku 2.1 je znázorněn průchod dat jednotlivými vrstvamy modelu ISO/OSI. Jak je z obrázku patrné tak ne všechny síťové zařízení pracují na stejných vrstvách.



Obrázek 2.1: Znázornění průchodu dat počítačovou sítí v modelu ISO/OSI

2.1.1 Fyzická vrstva / Vrstva síťového rozhraní

Nejnižší vrstva ISO/OSI modelu pracuje s daty na úrovni bitů a stará se o jejich přenos po přenosovém médiu. Protokoly této vrstvy definují signály, které reprezentují data a tudíž jde o protokoly implementované již v hardware síťových zařízení.

2.1.2 Linková vrstva

Linková vrstva je druhá nejnižší ISO/OSI modelu. Tato vrstva se stará o datovou komunikaci obecně mezi několika uzly, které jsou přímo spojeny. Spojení může být jak fyzickým vodičem tak i bezdrátovou technologií. Nejrozšířenější technologií pro fyzické spoje je Ethernet IEEE 802.3, pro bezdrátové spoje je to standard IEEE 802.11. Datová jednotka na linkové vrstvě se nazývá rámec a nese v sobě kromě zapouzdřených dat vyšších vrstev také informace o kontrolním součtu dat a adresování pomocí MAC adres. MAC adresa je adresa fyzického zařízení, které pracuje na této vrstvě. Adresování MAC adresou slouží pro identifikaci zařízení, které se nacházejí ve stejné počítačové síti a za hranici této sítě se již používá IP adresace, které je vysvětleno v necházející kapitole 2.1.3 Síťová zařizení pracující na této vrstvě se nazývaji switche. Úkolem switchů je zjistit MAC adresu cílové stanice a přeposlat je portem, který vede k tomuto zařízení.

2.1.3 Síťová vrstva

Na této vrstvě probíhá komunikace za využití IP adres. Prvky používané na této vrstvě jsou nazývaný routery. Účelem těcho zařízení je směrování paketů procházejících sítí. K tomu využívají směrovací tabulku. Právě pro problém vyhledání nejdelšího shodného prefixu v routovací tabulce směrovače jsou v této knihovně implementovány dva algoritmy, `Binary search on prefix length` **??** a `TreeBitmap` **??**. Datové struktury jsou pojmenované pakety. Pakety obsahují imformace o datovém toku a také zrojové a cílové adresy (IPv4 nebo IPv6) právě tohoto paketu.

2.1.4 Transportní vrstva

Transportní vrstva pracuje s datovou strukturou zvanou segmenty. Obsahují informace jako je kontrolní součet pro zjištění integrity dat, pořadové číslo rámce pro spojení dat, která byla na cestě k cíli rozdělena na více částí a také obsahuje čísla portů pro určení uživatelských aplikací, která data odeslala a která je na druhém konci má přijmout. Na transportní vrstvě se používají dva protokoly a to TCP a UDP. UDP má nižší režii ale zase je nespolihvé. To znamená že paket do cílového zařízení nemusí vůbec dorazit a nebo můžé dorazit v jiném pořadí než byl ze zdrojové stanice odeslán. Situace ve kterých pozitiva jako nižší režie přebyjí zápory je hlavně přenos dat v reálném čase. To je například streamování videa, přenost hlasu technologie VoIP a přenos informací do online her. Spolehlivý protokol TCPzaručuje, že všechna data budou přenesena a v cílové stanici se seřadí do stejné posloupnosti v jaké byla odeslána. Důvodem proč se data mohou rozdělit je nestabilní cesta, po které jsou data v síti směrována. Při přenosu jednoho toku dat se může stát že se jedna cesta po které se již poslala část dat stane nedostupnout a místo toho se data začnou směrovat přes jiná zařízení.

Zpracování dat na úrovni transportní vrstvy a všech vyšších vrstev není implementováno na síťových zařízeních starajících se přenost dat po síti. Jediná zařízení, které implementují zpracování dat těchto vrstev jsou koncová zařízení.

2.2 Časově kritické operace

Pod pojmem časově kritické operace se rozumí takové operace, které je typicky nutné provádět na více síťových zařízeních. Těmito zařizeními mouhou být routery, switche firewally

a také systémy oddělené od řízení síťového provozu jako například sondy pouze monitorující síťový provoz nebo analyzátory, které mohou hledat signatury útoků v datových tocích. Mezi časově kritické operace rozebrané v této práci patří klasifikace paketů a velký důraz je kladen specielně na jednodimenziální klasifikaci dle cílové IP adresy, vyhledávání nejdelšího shodného prefixu. Tato operace je využívána pro směrování na routerech. Dalšími z operací je hledání podřetězců a hledání regulárních výrazů. Poslední dvě zmíněné operace slouží především pro detekci útoků v systémech IDS (intrusion detection system) a pro prevenci útoků v systémech IPS (intrusion prevention system).

Co zmínit - co jsou tyto operace - na čem staví algoritmy - proč jsou kritické - zvyšování požadavků - rychlost zpracování - počet instrukcí při různých rychlostech

2.2.1 Hledání nejdelšího společného prefixu - Longest prefix matching

Problém hledání nejdelšího shodného prefixu se rozumí klasifikace paketů dle jejich cílové IP adresy, která může být jak verze 4, tak verze 6.

Tato operace je základním stavebním kamenem v počítačových sítích a bez ní by bylo téměř nemožné dosahovat efektivity jaké se dosahuje teď. Hledání nejdelšího shodného prefixu je operace, která se provádí na síťových prvcích zvaných směrovače. Tyto prvky jsou umístěny na každém rozhraní dvou a více počítačových sítích. Jejich cílem je nalézt nejvhodnější cestu, kterou směrovat příchozí paket. Struktura reprezentující uložené směrovací informace se nazývá routovací tabulka. Tato tabulka ukládá informace o dostupných sítích (jejich prefixech), délce tohoto prefixu a rozhraní, kterým se lze do odpovídající počítačové sítě dostat. S velkým rozmachem počíačových sítí v poslední dekádě dochází k velkému nárustu routovací informací a proto se tímto směrem začal ubírat akademický výzkum. Pro zmenšení routovacích tabulek byl navržen takzvaný supernetting, který sdružuje sítě se stejným prefixem o jiných délkách a stejnou cestou do jedné sítě. Tím je dosaženo zmenšení routovacích tabulek, nicméně i tak je potřeba v routovacích tabulkách vyhledávat efektivně.

Nejdelší shodný prefix se zapisuje jako $1001*$ a tento zápis reprezentuje všechny IP adresy, které v binární podobě začínají právě na hodnotou 1001. Nicméně v routovací tabulce může být uložen i prefix $10010*$, který sdílí první čtyři bity své adresy s výše uvedeným příkladem, a případě, že přijde paket začínající hodnotou 10010 je z pohledu směrování vyhodnotit prefix B jako nejdelší a poté paket správně směrovat. V případě 10011 je však nejdelším shodným prefixem pravidlo A a tudíž nesmí dojít k vyhodnocení prefixu B jako nejdelšího. Z toho důvodu je nutné ve směrovací tabulce uchovávat i informace o délce prefixu. Tato informace poté slouží pro rozhodnutí, jaký pravidlo routovací tabulky má nejdelší shodný prefix. Délká prefixu může nabývat hodnot $1 - 32$ pro adresy typu IPv4 a $1 - 128$ pro adresy typu IPv6. Z výše uvedených tvrzení je zřejmé, že pro každý typ IP adres musí existovat samostatná, oddělená routovací tabulka, jinak by mohlo docházet k nevalidním vyhodnocení nejdelšího shodného prefixu mezi IPv4 a IPv6 adresami. Jako příklad může sloužit následující situace

100* 3 R1 IPv4 100* 3 R7 IPv6

Teoreticky by bylo možné ukládat různé typy prefixů do stejných datových struktur pro algoritmus Binary search on prefix length, nicméně by bylo nutné explicitně rozlišovat o jaký typ IP adresy se jedná a to by mělo negativní vliv na rychlost vyhledávání. To je právě kritický paramter, který se snažíme minimalizovat.

Pro hledání nejdelšího shodného prefixu existuje velké množství algortimů, které jsou popsány v [] Většina z nich je založena na procházení stromové struktury. Každý algorit-

mus má jiné paměťové nároky a dosahuje jiných rychlostí. Je nutné zvolit kompromis mezi rychlostí a paměťovou náročností. V případě že jde o implementaci na architektuře FPGA důraz bude pravděpodobně kladen na paměťovou náročnost a to z důvodu, že tyto čipy mají specializované paměti. Tady bych zmínil asociativní paměť, kterou lze přímo použít pro vyhledávání. Na architekturách vycházejících z x86 je naopak kladen důraz na rychlost zpracování z důvodů obecných procesorů, které nejsou specializovány na tyto operace ale naopak zase disponují standardními paměťmy, které jsou lehce škálovatelné a dosahují kapacit řádově převyšující nároky jednotlivých algoritmů.

Algoritmy rozehrané v této kapitole vycházejí z obecně n-árního stromu, což je rozšíření binárního stromu na více než dva potomky. Těmito algoritmy jsou `Binary search on prefix length`, jehož data jsou uložena v binárním stromu, ale pro vyhledávání je využito výhod hašovací tabulky, a `TreeBitmap`, který je n-árním stromem, kde n je volitelné. Dosažené výsledky pro různé n jsou vizualizovány v kapitole **??**.

První algoritmus hledání nejdelšího shodného prefixu byl založen na naivním procházení lineárního seznamu a byl publikován v knížce X v roce Z. Jak si čtenář může dovtípit do vyhledání byla závislá na počtu uložených prefixů a její časová složitost byla $O(N)$. Algoritmy popsané a implementované v rámci této práce se minimalizují dobu vyhledávání v závislosti na počtu prefixů v routovací tabulce. V případě binary search on prefix length je tato závislost omezená vlastnostmi hašovací funkce. Pokud je zvolena špatná hašovací funkce bude docházet ke kolizím a v případě nefunkční hašovací funkce (generuje stejný výsledky pro řůzné prefixy) ořezán na prosté procházení lineárního seznamu.

Co zmínit - jaké jsou známené algoritmy - hašovací funkce - výhody/nevýhody těchto algoritmů - první algoritmus - směrovací protokoly - co obsahuje směrovací tabulka - jedna adresa může odpovídat více záznamům - příklad směrovací tabulky s stejnými adresami a různou délkou - délka je počet bitů adresy sítě - umísťění zařízení má vliv na velikost tabulky - nějaký příklad? - sekvenční vyhledávání v seznamu - TCAM, asociativní paměť? - trie struktura? - binární trie - multibitový trie - klíč je určen implicitně pozicí ve stromu a ne explicitně

TreeBitmap

Struktura pro uložení prefixu se skládá z položek interní bitmapy, externí bitmapy, ukazatele na potomky a ukazatele na pravidla. Interní bitmapa zahrnuje všechny všechny prefixy, které je možné vyjádřit na N bitech střídy. Jako příklad může posloužit velikost střídy 2 bity. Proto je nutné uložit všechny tyto kombinace: $0*$, $1*$, $00*$, $01*$, $10*$ a $11*$. Z toho lze odvodit vzorec pro počet bitů nutných pro reprezentaci všeh možný střídy. 2^{N+1} kde N je velikost střídy. Velikost externí bitmapy lze vyjádřit vzorcem 2^N kde N je velikost střídy.

Interní bitmapa slouží pro zjištění, kolik pravidel existuje v tomto uzlu. To se zjistí zavolání funkce `ones(internal-bitmap, end)`. kde end pozice, která odpovídá části prefixu o délce střídy. To znamená že pokud po vykousnutí střídy z prefixu dostaneme bity 11 pozice end bude stanovena na decimální reprezentaci tohoto čísla, v tomto případě 3. Následuje zjištění, zda se na dané pozici interní bitmapy vykytuje jednička nebo nula. V případě, že je tam nula se neděje nic a algoritmus pokračuje zjišťováním, zda existuje cesta stromem dál z externí bitmapy jak je vysvětleno v následujícím odstavci.

Pokud je tam jednička tak algoritmus spočítá funkci ones na interní bitmapě a uloží si vypočítanou hodnotu. Dále si také uloží odkaz do tabulky pravidel.

Pokud tam není jednička, tak se postupně odebírají méně významné bity vykousnuté v

Prefix	$*$	$0*$	$1*$	$00*$	$01*$	$10*$	$11*$
Pravidlo	0	0	1	0	0	1	1

Tabulka 2.1: Příklad interní bitmapy

Prefix	$00*$	$01*$	$10*$	$11*$
Pravidlo	1	0	1	0

Tabulka 2.2: Příklad externí bitmapy

prefixu a v cyklu se provádí zjišťování zda je tam jednička nebo ne. Tento cyklus je ukončen v případě, že se narazí na nultou pozici nebo je nalezena jednička.

V případě že již neexisuje další cesta stromem je algoritmus vyhledávání ukončen a jako výsledek je navrácena hodnota uložená ukazateli na nejlepší pravidla s index, který je uložen také.

Externí bitmapa slouží pro zjištění, zda existuje cesta ve stromu, která odpovídá vyzobnutému prefixu. Index do pole následovníků je vypočítán jako `ones(external-bitmap, end)`. V případě, že pro vyzobnuté bity neexistuje následovník je navráceno pravidlo, které bylo zjištěno z interní bitmapy.

Funkce `ones(bitmap, end)` spočítá všechny bity, které jsou nastaveny na jedničku v rozmezí $< 0, end)$. Toto je implementuje jako bitová operace `AND` a maskou, která reprezentuje právě počet pozic, které se mají vypočítat. Samotné zjištění počtu jedniček lze řešit několika způsoby, v implementaci této knihovny je zvoleno volání vestavěné funkce překladače `popcount`, které se při překladu převede na nejefektivnější instrukce cílové platformy.

Algoritmus v pseudokódu je možné vidět v 1.

Příklad bitmap pro uložení prefixů:

Treebitmap je šetrný k paměti a alokuje si pouze tolik prvků pole ať už pro uložení pravidel nebo pro uložení následovník kolik jich je opravdu potřeba.

Z [3] vyplívá, že je vhodné nastavit velikost střídy v rozmezí 2 - 8 bitů. Což je také rozmezí, na které je sousředěna a otestována implementace TreeBitmap v této knihovně. Experimentálně bylo zjišténo, že tato implementace je funkční až do velikost střídy 13bitů.

Data: tbm-root, ip, ip-length
Result: routing rule
node ← tbm-root;
position ← 0;
repeat
  bits ← get-stride-bits(ip, position, prefix-length);
  position ← position + STRIDE;
  if *isRule(node.internal, bits)* **then**
    longest-match = node;
  end
  index ← ones(node.external, bits);
  parent ← node;
  node ← node.external[index];
until *BIT(parent.external, bits)*;
return longest-match;

Algorithm 1: Hledání nejdelšího shodného prefixu algoritmem TreeBitmap

Co zmínit - procházení interních a externích bitmap, struktura interní a externí bitmapy - alokace všech násldovníků a pravidel na jednou v jednom místě - funkce jedniček

Binary search on prefix length

Algorimus binary search on prefix length je založen jak již název napovídá na binárním vyhledávání. Binaární vyhledávání jiný název pro půlení intervalů. Vyhledávání půlením intervalů se ve své první iteraci pokusí vyhledat shodný prefix celé délky, tedy 32 bitů pro IP adresu verze protoklou IPv4 a 128 bitů pro IP verze 6.

Mějme směrovací tabulku obsahující tyto informace:

Prefix	Délka prefixu	Pravidlo
147.228.0.0	14	P1
147.228.128.0	17	P2

Tabulka 2.3: Příklad směrovací tabulky

Vyhledávání směrovací cesty pro adresu 147.229.128.54 bude procházet následujícími kroky

Prefix	Délka prefixu	Změna prefixu	Průběžný výsledek
147.228.128.54	32	16	Nenalezeno, zmenšit délku
147.228.0.0	16	8	Nenalezeno, zvětšit délku
147.228.128.0	24	4	Nenalezeno, zmenšit délku
147.228.128.0	20	2	Nenalezeno, zmenšit délku
147.228.128.0	18	1	Nenalezeno, zmenšit délku
147.228.128.0	17	0	Nalezeno, vrátit P2

Tabulka 2.4: Příklad vyhledání nejdelšího shodného prefixu

Operace vyhledání nejdelšího prefixu při využití algoritmu binary search on prefix length má časovou složitost $\log 2N$, kde `N` je počet bitů adresy. V případě IPv4 adresy je to 32 bitů a pro IPv6 adresu je to 128 bitů. Z principu algoritmu vyplívá, že nejhorší výsledky z časového hlediska bude dosahovat při shodě prefixu, který byl zadán s lichou délkou. V tomto případě bude nutné projít všemy kroky. Počet kroků v případě IPv4 bude 5 a v případě IPv6 adresy to pak bude 7. Zde je vidět že i v připadě čtyřikrát delší adresy se počet kroků pro vyhledání prefixu zvedne pouze o dva, což neplatí pro algoritmus TreeBitmap, který musí projít v nejhorším případě až čtyřikrát více uzlů aby nalezl odpovídající prefix. [2]

Co zmínit - leaf-pushing?

2.2.2 Hledání řetězců

V počítačových sítích je často vyžadováno hledání řetězců v síťovém provozu ať už z důvodu monitorování nebo blokování provozu, který obsahuje určité řetězce. Blokování může probíhat na úrovni obsahu, například blokování určitých webových stránek ať už z důvodů rodičovské kontroly nebo například omezení provozu netýkajícího se výuky ve školních sítích. Nebo může jít o blokování určitého druhu síťového provozu, který obsahuje nedovolené nebo podezřelé signatury. Pokud se vzdálí ze síťového provozu tak důvodem pro vznik těchto

```
Data: bspl-root, hash-table, ip, ip-length
Result: routing rule
prefix-length ← ip-length;
prefix-change ← ip-length;
repeat
    bits ← get-prefix-bits(ip, prefix-length);
    item ← hast-table.get(bits);
    prefix-change ← prefix-change ≫ 1;
    if item == NULL then
        prefix-length ← prefix-length - prefix-change;
    end
    else if item.type == PREFIX then
        prefix-length ← prefix-length + prefix-change;
    end
    else break;
until prefix-change > 0;
if item == NULL then return bspl-root.default-rule;
```

Algorithm 2: Hledání nejdelšího shodného prefixu algoritmem Binary search on prefix length

algoritmů je hledání v textových datech. I z tohoto důvodu vznikl algoritmus Aho-Corasick, který je v této knihovně implementován a původně byl navržen právě pro hledání klíčových slov v odborných publikacích.

Pro hledání řetězců je implementován algoritmus autorů Aho a Corasicové. Tento algoritmus používá pro zjištění shody s podřetězcem konceptu konečného automatu. Při každé iteraci algoritmu se provede přechod o jeden znak.

Algoritmus procházení vstupních dat je rozepsán v 2.2.2 a vychází z [1].

```
Data: start-state, text
Result: keyword
state = start-state;
for position ← 0 to text.length do
    while goto(state, text[position]) == FAIL do state ← state.failure;
    if state.isMatch then return state.keyword;
end
return NOT-MATCH;
```

Algorithm 3: Algoritmus procházení textu a hledání podřetězců

Při chybě alokace se smaže celá struktura konečeného automatu, neexistuje předpoklad na dynamické přidávání pravidel za běhu programu

2.2.3 Hledání regulárních výrazů

Regulární výrazy slouží pro popis operací nad jazykem. Jazek je definovaný jako iterace nad vstupní abecedou. Iterace může být neutrální nebo kladná. V neutrální iteraci jazyka je oproti pozitivní iteraci zahrnut i symbol ϵ, který reprezentuje prázdný znak/řetězec. Iterace jazyka se označuje jako L^n kde `n` je označení iterace jazyka. iterace L^0 obsahuje

pouze jeden symbol a tím je ϵ ϵ také reprezentuje prázdný řetězec. Prázdný řetězec se může skládat z nekonečné posloupnosti ϵ Vstupní abeceda je množina symbolů.

Regulární výrazy jsou poté operace nad regulárními jazyky. Regulární výrazy mají stejnou vyjadřovací schopnost jako konečné automaty a pro digitální zpracování regulárních výrazů se vždy používají konečné automaty.

Operace regulárních výrazů jsou následující

- $\emptyset$ je regulární výraz reprezentující prázdnou množinu
- ϵ je regulární výraz reprezentující $\{\epsilon\}$
- $a, a \in \Sigma$ je regulární výraz reprezentující $\{a\}$
- $(r \cdot s)$ je regulární výraz reprezentující RS
- $(r|s)$ je regulární výraz reprezentující $R \cup S$
- $(r*)$ je regulární výraz reprezentující R*

Znak operace konkatenace se čast vynechává a je uvažován implicitně.

Regulární výrazy implementované v této knihovně rozšiřují množinu operací tři nové druhy zápisu, které jsou pouze pohodlněji zapsatelná a nijak nerozšiřují výrazové možnosti regulárních výrazů.

- $[abc]$ je výčet znaků, které se na vstupu mohou vyskytnout a automat je v aktuální stavu dokáže zpracovat. Je to zkrácený tvar zápisu $(a|b|c)$
- $a+$ je definováno jako pozitiní iterace, tedy $1..N$ opakování
- $a?$ je definováno jako 0..1 iterací

Regulární jazyky se podobně jako **??** používají pro monitorování a blokování dat procházejících počítačovými sítěmi. Zařizení využívající těchto operací jsou typicky firewally, a to jak hardwarové tak i jejich softwarové implementace.

Při chybě alokace se smaže celá struktura regulárního výrazu, neexistuje předpoklad na dynamické přidávání regulárních výrazů za běhu programu.

2.2.4 Analýza a extrakce hlaviček paketů - Header parsing

Extrakce hlaviček paketů je velmi častá operace. V případě hledání nejdelšího shodného prefixu je potřeba nejprve provést extrakci cílové adresy a až poté je možné zahájit vyhledávaní cesty, kterou bude packet směrován. Z toho vyplívá přímá závislost rychlosti všech operací pracující s položkami hlavičky jak paketů na úrovní síťové vrstvy 2.1.3 tak i rámců na úrovni linkové vrstvy 2.1.2 a segmentů na úrovni transportní vrstvy 2.1.4.

V rámci výzkumu byly navrženy způsoby hardwarové akcelerace na zařízeních typu FPGA **??** na publikaci z ANT.

V rámci LPM je stačí extrahovat pouze jednu položku IP hlaviček, nicméně v obecné klasifikaci paketů jsou potřeba i další položky jako zdrojová adresa, cílový port, typ transportního protokolu.

Kapitola 3

Návrh API knihovny

Knhovna `fastnet` je navržena jako množina menších knihoven, kde každá knihovna implementuje jednu operaci používanou při zpracování síťového provozu. Tímto návrhem je dosaženo snadné rozšiřitelnosti o další operace, jako například extrakce informací z hlaviček paketů ıpacker header extraction nebo klasifikace paketů `packet clasification`. Mezi implementované operace patří vyhledání nejdelšího shodného prefix `longest prefix match ??`, hledání podřetězců `pattern matching` a regulární výrazy `regular expressions`. Pro vyhledání nejdelšího shodného prefixu jsou implementovány algoritmy `Binary search on prefix length??` a `Tree Bitmap??`. Hledání podřetězců je implementováno algoritmem `Aho-Corasick??`. Regulární výrazy jsou řešeny nedeterministickým i deterministickým konečným automatem.

Další výhodou tohoto rozdělení je možnost snadno vytvořit a používat jednotlivé podknihovny samostatně. To se může hodit pro zařízení, která mají velmi limitované paměťové úložiště a jejich účelem je řešit pouze jednu ze zmíněných operací.

Veřejné rozhraní knihovny se skládá z veřejných rozhraní jednotlivých podknihoven. Tyto rozhraní jsou popsány v následujících podkapitolách.

3.1 Vyhledání nejdelšího shodného prefixu

Pro operaci vyhledání nejdelšího shodného prefixu jsou připraveny funkce pro inicializace datových struktur `lpm_init`, pro vložení nového prefixu `lpm_add`, aktualizaci existujícího prefixu `lpm_update`, smazání existujícího prefixu `lpm_remove` a zrušení všech alokovaných datových struktur `lpm_destroy`. Jak již bylo zmíněno je nutno uchovávat zvlášť tabulky pro IPv4 a IPv6 adresay a z toho důvodu jsou existují všechny výše uvedené funkce i ve variantě pro IPv6. Jediným rozdílem pak je že je všechny funcke a datové struktury mají místo prefixu `lpm_` prefix `lpm6_`.

Všechny funkce kromě `lpm_init` pracují s paramterem typu `lpm_root`, který reprezentuje celou datovou strukturu. Tato implementace vychází z požadavku mít možnost využívat více routovací tabulek nebo tyto struktury využít pro jiné typy klasifikace.

Funkce pro vložení, smazání a aktualizci pravidel a prefixů také obsahují parametr prefix, který je buď IPv4 nebo IPv6 adresa. Dalším parametrem těchto funkcí je délka prefixu, aby bylo možné odlišit jednotlivé prefixy od sebe. Tam může dojít ke kolizi pokud existuje prefix $1*$ o délce jedna a prefix $10000*$ o délce pět, neboť tyto prefixu budou reprezentovány stejným číslem, neboť v binární kódu máme pouze dvě hodnoty a není možné určit třetí stav jako X, nezajímá nás/není důležité.

Funkce pro vložení prefixu pracuje s pouze s jedním prefixem a je prováděna okamžitě. Tento návrh vychází z předpokladu, že knihovna bude používána i v prostředí s dynamickými routovacími protkoly jako například RIP, OSPF nebo BGP, které při změně směrovacích informací v případě OSPF (zjistit jak to je) zasílají aktualizace s novými informacemi na všechny sousední routery nebo jsou tyto změny zasílány periodicky v případě protokolu RIP.

Hašóvací funkce je v této implementaci použita henkins **??**. Volba správné hašovací funkce má má velký vliv na rychlost vyhledávání. V případě kolizí je to omezeno zase na procházení lineárního seznamu a z časové složitosti $O(\log 2N$, kde N reprezentuje délku adresy, tedy 32 nebo 128 bitů, se stává $O(N)$, kde N reprezentuje počet záznumů routovací tabulky.

Přesnou specifikaci rozhraní je možné nalézt v přiloženém CD ve složce `/lib/src/lpm/lpm.h`

3.2 Hledání řetězců

Pro hledání řetězců jsou podobně jako pro vyhledání nejdelšího shodného prefixu implemtovány funkce pro inicializaci datové struktury `pm_init`, vložení klíčových slov do dané struktury `pm_add`, změnu pravidla odpovídající danému klíčovému slovu `pm_update`, smazání klíčového slova `pm_remove` a funkce pro uvolnění paměti alokované pro vyhledávání `pm_destroy`.

Všechny výše zmíněné funkce očekávají jako první parametr strukturu typu `pm_root`, která je základním prvkem pro vyhledávání a právě do této struktury jsou uložena všechna klíčová slova a jejich pravidla. Důvodem proč tato struktura není globání proměnnou uvnitř knihny je požadavek na možnost mít více prohledávacích struktur a mít možnost explicitně zvolit oproti jaké struktuře klíčových slov se budou vstupní data porovnávat.

Hledání podřetězců `pm_match` skončí svůj průchod konečný automatem v momentě nálezu první shody s libovolným podřetězcem zadaným při volání `pm_add`. Vpřípadě, že není nalezena žádná shoda se vstupními podřetězci je vrácen výsledek `false`.

Čtvrtým parametrem funkce `pm_match` může být `NULL` nebo odkaz na datovou strukturu `pm_result`. V případě `NULL` argumentu již nelze procházet textem a hledat další shody. Pokud je zadán odkaz na existující strukturu `pm_result` je možné procházet celým textem a ukládat všechny nalezené shody s podřetězci. Jednotlivé položky struktury `pm_result` jsou.

Pro uložení výsledků procházení textu je možné používat různé proměnné typu `pm_result` a je jen na programátory konkrétní aplikace jaký postup zvolí.

Pro práci se strukturou `pm_result` jsou v knihovně `pm` implementovány následující operace: `pm_result_init` pro vytvoření této struktury a `pm_result_destroy` pro uvolnění paměti alokované pro tuto strukturu. Položkami této struktury jsou údaje o pozici v textu, kde bylo nalezena klíčová slova uložená v položce `rule` o velikost `count`, aktuální stav, ve které se konečný automat nachází a to z důvodu možného volání `pm_match_next`, kdy vyhledávání naváže na místě kde předchozí vyhledávání skončilo.

Pro vkládání podřetězců je vytvořena vlastní datová struktura, a pole těchto struktur je předáváno do funkce `pm_add`. Důvodem pro tuto implementaci namísto přidávání jednotlivých podřetězců samostatně je relativně časově náročné procházení stavového automatu a generování tzv. failure přechodů, které je nutné provést po každé změně datové struktury.

Funkce `pm_add` očekává jako druhý parametr pole struktur `pm_keyword`, kde každá struktura obsahuje položky vstupního klíčové slova v binární podobě, délku tohoto slova a pravidlo odpovídající tomuto slovu, oproti kterým se budou porovnávat vstupní data. Důvodem

proč je předáváno pole dat a ne jednotlivé položky jako u hledání nejdelšího shodného prefixu je z důvodu neexistence synchornizační mechanismů v prostředí IPS a IDS. Dalším důvodem pro tento návrh je časově náročná funkce generování tzv. failure přechodů, které umožňují detekovat kratší klíčové slovo i v případě, že již je zahájeno provnávání delšího slova jak je možno vidět na následujícím příkladu.

3.3 Regulární výrazy

Knihovna `regex` implementující regulární výrazy nabízí dvě možnosti pro procházení vstupních dat a to deterministický `dfa` a nedeterministický konečný automat `nfa`. Z toho důvodu jsou odlišeny všechny funkce dle typu konečného automatu, který je použit. Oproti hledání nejdelšího shodného prefixu **??** nejsou tyto implementace od sebe odlišeny různými soubory neboť deterministické automaty vycházejí z nedeterministický a pouze používají determinizaci.

Pro nedeterministické konečné automaty to jsou následující funkce.

```
regex_nfa * regex_construct_nfa(regex_pattern patterns[], unsigned count);
int regex_match_nfa(regex_nfa * root, char * input, unsigned length);
void regex_destroy_nfa(regex_nfa * root);
```

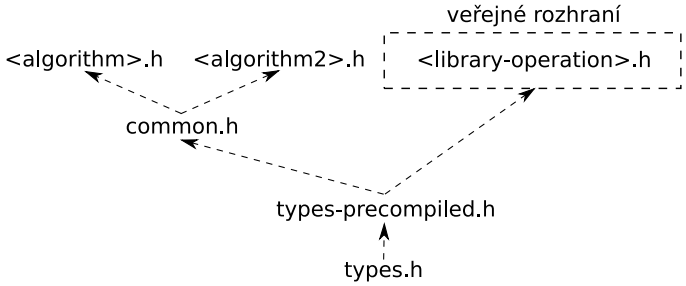
Pro deterministické konečné automaty jsou to tyto funkce.

```
regex_dfa * regex_construct_dfa(regex_pattern patterns[], unsigned count);
int regex_match_dfa(regex_dfa * root, char * input, unsigned length);
void regex_destroy_dfa(regex_dfa * root);
```

Pro vytvoření regulárních výrazů je podobně jako u hledání podřetězců **??** použita pomocná struktura `regex_pattern`, která je předávána do funkcí `regex_[nfa|dfa]_construct`. Výsledný konečný automat, ať už deterministický nebo nedeterministický je výsledek spojení jednotlivých konečných automatů pro každý regulární výraz. Tím je umožněna detekce shody několika regulární výrazů v jednom průchodu vstupními daty i s přesnou identifikací jaký regulární výraz se shoduje se vstupními daty. To je také jedním z důvodů proč nejsou použity regulární výrazy ze standardní knihovny jazyka C. Další důvody pro vlastní implementaci regulární výrazů je možnost zvolit nejvhodnější algoritmus pro danou operaci. Mějme na paměti že determinizací může dojít až k exponenciální nárustu stavů konečného automatu, což může problém zvláště při použití specializovaných HW implementací, kde je možno narazit na paměťové limity. Dalším důvodem je převod implementace do akcelerovaného hardware.

```
typedef struct
{
        unsigned length;
        unsigned char id;
        char * input;
} regex_pattern;
```

hlavičkové soubory jsou rozděleny na veřejné a privátní rozhraní, kde privátní rozhraní je používáno pouze uvnitř knihovny. Hierarchickou strukturu je možné vidět na obrázku 3.3.



Obrázek 3.1: Diagram závislostí hlavičkových souborů

Jako výchozí hlavičkový soubor je použit types.h, který obsahuje definice datových struktur pro všechny algoritmy v podknihovně, které musí být viditelné i z veřejného rozhraní. Dalším souborem je types-precompiled.h, který je generován z types.h při překladu když se vybírá používaný algoritmus. common.h je hlavičkový soubor společný pro všechny algoritmy v podknihovně a algortihm.h pak obsahuje deklarace právě pro jeden konkrétní algoritmus. sublib.h je pak hlavičkový soubor, který tvoří veřejné rozhraní ke knihovním funkcím.

3.4 Použití knihovny

Celou knihovnu je možné sestavit příkazem `make all` v hlavním adresáři knihovny. Při tomto příkazu bude celá knihovna sestavena s definovaným NDEBUG, což má za následek vypuštění všech volání funkce assert, která slouží pro ověřovaní správné funkčnosti.

Dalšími cíly pro program `make` jsou

- `test` - spustí automatické testování všech částí knihovny
- `bech` - spustí benchmarky všech částí knihovny
- `clean` - smaže všechny soubory vytvořené překladem

3.5 Rošíření knihovny

Pro rozšíření knihovny je nutné přidat knihovnu implementující danou operaci do adresáře `lib/src` a upravit příslušný soubor `Makefile` v daném adresáři. Dále je vhodné vytvořit testovací program a sadu testů, kterou je možné automatizovaně spouštět a vyhodnocovat. Tyto soubory pak umístit do adresáře `lib/test/<operace>`. Další vhodnou součástí knihovny operace je benchmark pro vyhodnocení rychlosti/paměťové náročnosti jednotlivých implementací dané operace.

3.5.1 Packer header extraction

V knihovně je již navrženo API pro operaci extrakci informací z hlaviček paketů.

`pc_set` je struktura typu union, kde jsou všechny položky uloženy na stejném paměťovém místě a jejich interpretace je odvozená dle typu položky, ke které je přistoupeno.

```
typedef union
{
```

```
        uint32_t number;
        uint16_t number16;
        char character;
        in_addr addr;
} phe_item;

typedef union
{
        uint32_t number;
        uint16_t number16;
        char character;
        in6_addr addr;
} phe6_item;

_Bool phe_get(char * input, phe_item * items, ...);
_Bool phe6_get(char * input, phe6_item * items, ...);
```

3.5.2 Packer clasification

V knihovně je již navrženo API pro operaci klasifikace paketů.

```
typedef struct
{
        unsigned rule;
        struct in_addr dst;
        struct in_addr src;
        _Bool protocol;
        short port;
} pc_set;

typedef void pc_root;

pc_root * pc_init();
void pc_destroy(pc_root * root);
_Bool pc_add(pc_root * root, pc_set set[], unsigned count);
_Bool pc_update(pc_root * root, pc_set old, pc_set new);
void pc_remove(pc_root * root, pc_set set);
```

Kapitola 4

Výsledky

Tato kapitula shrnuje a vizualizuje dosažené výsledky při implementaci jednotlivých knihoven. Benchmarky pro architekturu Intel proběhly na operačním systému Archlinux [1] s procesorem Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz.

4.1 Hledání nejdelšího shodného prefixu

Hledání nejdelšího shodného prefixu bylo testováno na celkem pěti vstupních sadách dat, které byly převzaty z volně dostupných dat RIPE[2]. Informace o testovaných sadách dat jsou zobrazeny v tabulce 4.1.

Tabulka 4.1: Informace v routovací tabulce pro benchmarky

Počet adres	Nejkratší prefix	Nejdelší prefix	Verze IP
1000	13	31	IPv4
10000	8	32	IPv4
100000	8	32	IPv4
1000	23	128	IPv6
10000	19	128	IPv6

Vyhledávání pak bylo testováno oproti sadě velikost 1000 odpovídající verze IP, která je podmnožnou každé rozsáhlejší sady dat.

Testováno bylo jak TreeBitmap ve všech variantách velikosti střídy, tedy $1 - 8$. Jak je vidět na následujících grafech, tak nejlepších výsledků bylo dosaženo pro TreeBitmap s velkostí střídy nastavenou na 5 bitů a to jak pro IPv4 tak i pro IPv6.

Pro testování algoritmu Binary search on prefix length byla zvolena velikost hešovací tabulky stejná jako velikost směrovacích dat. Tento přístup byl zvolen z důvodu významného vlivu veliskosti hešovací tabulky na rychlost samotného vyhledávání. V nejhorším případě, tedy při velikosti hešovací tabulky 1 byl algortimus pomalejší v až o U algoritmu bspl je doba vyhledání prefixu velice závislá na velikosti hešovací tabulky a proto je vhodné odhadnout počet záznamů tabulky alespoň řádově a dle toho pak nastavit velikost konstantu `_HTABLE_SIZE` v souboru bspl.h na hodnotu, která alespoň řádově odpovídá předpodkládané velikosti hešovací tabulky.

[1] https://www.archlinux.org/
[2] https://www.ripe.net/

Dalším důležitým faktorem pro efektivnost implementace je velikost datových struktur, která je znázroněna v následující tabulce.

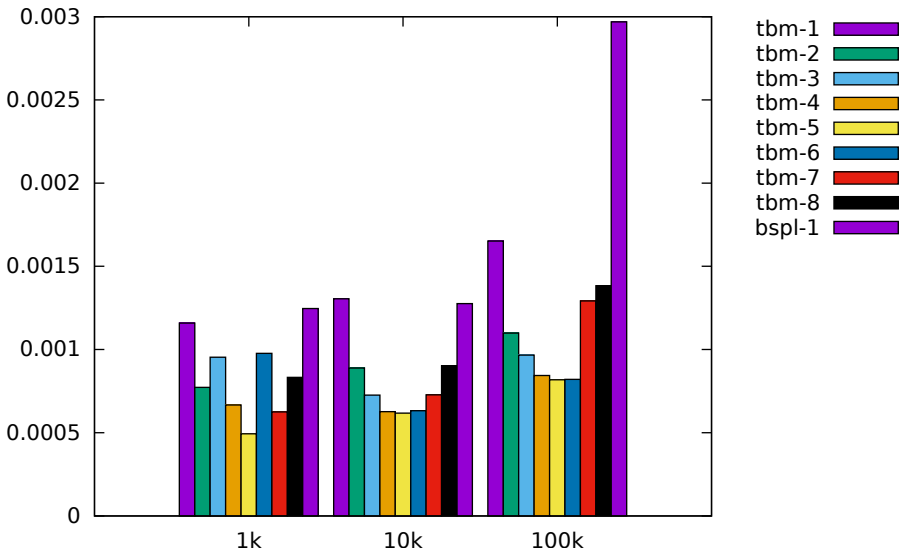
Tabulka 4.2: Velikosti datových struktur pro TreeBitmap

Velikost střídy	počet bytů
1	24
2	24
3	24
4	24
5	32
6	40
7	64
8	112

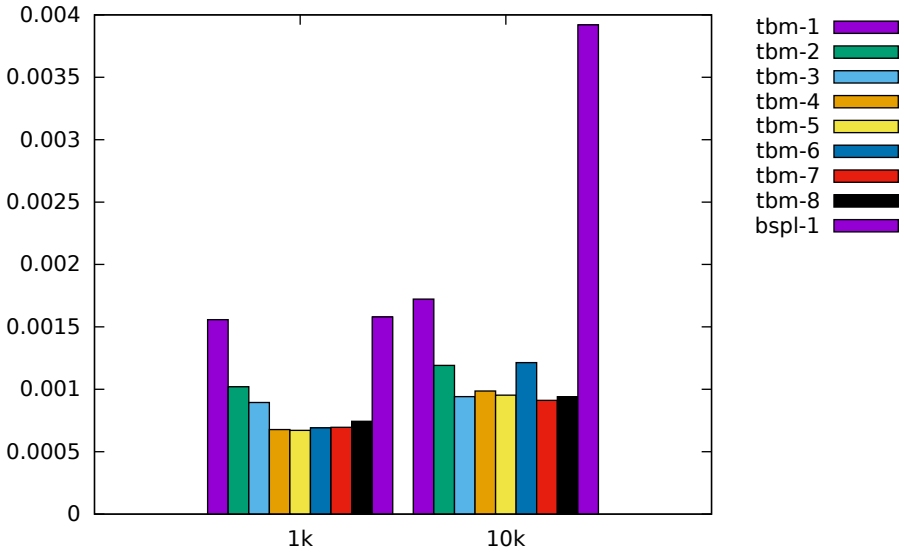
Pro Binary search on prefix length pak bude velikost jednoho prvku stromu činit $48B$. Pokud by došlo k rozdělení datových struktur pro jednotlivé verze IP protokolu tak verze pro IPv4 by měla velikost $36B$.

Jak je vidět ve výše uvedené tabulce tak i přestože nejrychlejší implementací je TreeBitmap s střídou 5, tak v případě omezené paměti by bylo vhodnější zvolit kompromis mezi rychlostí a paměťovou náročností v podobě TreeBitmap s velikostí střídy 4. Dalším ovlivňujícím faktorem je odhadovaný počet záznamů směrovací tabulky, protože nejrychlejší TBM-5 při existenci všech struktur na všech úrovních bude činit rozdíl zabraného místa oproti TBM4 celých $TODOdopotatKB$, což je v kontextu FPGA případně ASIC implementace nezanedbatelné množství.

Také je důležité zmínit že z počtu očekávaných vstupních pravidel má také vliv na velikost struktury. Do 256 záznamů zabírá pravidlo pouze 1byte, do 65536 záznamů pak 2 byty a pro více než 65536 pak celé čtyři byty pro uložení právě jednoho pravidla. Při uložení 65536 adres to bude rozdíl mezi uložením 65537 bude rozdíl činit $200KB$



Obrázek 4.1: Benchmark pro IPv4



Obrázek 4.2: Benchmark pro IPv6

Kapitola 5

Závěr

Cílem této práce bylo popsat a navrhnout aplikační programové rozhraní časově kritické operace v oblasti počítačových sítí, konkrétně vyhledání nejdelšího shodného prefixu za využití algoritmů Binary search on prefix length a TreeBitmap. Operace používané na směrovačích pro zjištění jakou cestou směrovat přicházející pakety. Dalšími operacemi jsou hledání řetězců a hledání regulárních výrazů v paketech a jejich datech. Pro operaci hledání řetězců je to konkrétně algoritmus Aho-Corasick [1] umožňují v při jednom průchodu vstupními daty vyhodnit zda se v datech nacházejí specifikovaná klíčová slova a poté hledání regulárních výrazů. Obě zmíněné operace jsou používané v paketových filtrech, které tvoří jádro systémů pro detekci útoků (IDS) a prevenci útoků (IPS). Do skupiny těchto systémů patří firewally jak softwarové tak hardwarové. Mezi další operace operace pak patří analýza a extrakce hlaviček paketů a obecná klasifikace paketů.

Dále je to pak implementace operací hledání nejdelšího shodného prefixu, hledání řetězců a hledání regulárních výrazů. Při implementaci bylo dosaženo rychlosti zpracování N paketů za sekundu při velikosti 100000 záznamů routovací tabulky pro IP adresy verze 4 a N paketů/sekunda při velikosti routovací tabulky 10000 pro IP adresy verze 6. Pro operaci hledání podřetězců bylo do dosaženo rychlosti zpracování N paketů za sekundu v závislosti na počtu shodných písmen při lineární náročnosti v závislosti na velikost dat v paketech. Experimentování bylo prováděno na vzorku dat odchycených ze standardního síťového provozu jednoho osobního počítače. Jako vzorek testovaných klíčových slov bylo využito klíčových definovaných pro HTTP[1]. Pří experimentování s regulárními výrazy byly jako vstupní data použita stejná data jako pro hledání řetězců, ale jako regulární výraz byly použity matchování URL[2] adres.

Část této práce pojednávající o hledání nejdelšího shodného prefixu vychází z publikací [3], [2], a dalších. Hledání řetězců vychází z [1] a hledání regilárních výrazů z [].

5.1 Další rozšíření

Jako kroky navazující na tuto práci je možné implementovat zbývající operace, které implemetnovány nebyly. Těmito operacemi je klasifikace paketů a analýza a extrakce hlaviček paketů.

[1] Hyper-text transfer protocol
[2] Uniform resource locator

5.1.1 Optimalizace

U implementace `Binary search on prefix length` je možné rozdělit strukturu `_bspl_node` na dvě a to jednu pro každou verzi IP protokolu. Tím se dosáhne snížení paměťové náročnosti pro implementaci IPv4 o 12B pro každý uzel. To s sebou nese nutnost upravit všechny funkce pracující s položkou `prefix` v rámci této struktury.

Další optimalizací je přepsání leaf-pushing do iterativní průchod za použití Morrisova algoritmu [4], který umožňuje průchod stromovou strukturou bez použití rekurze a zásobníku.

Dále je možné spojit položky `type` a `prefix_length` struktury `_bspl_node` do jednoho byte. To je možné z důvodu rozsahu `prefix_length` $1 - 128$ a pouze dvou typů uzlu, což je možné reprezentovat jedním bytem.

5.1.2 Stress-testing

Ze specifikace požadavků na implementaci knihovny v nízkoúrovňovém jazyce C je zřejmé, že knihovna bude používána i na vestavěných systémech disponujících omezenou pamětí a z toho důvodu by jedním z dalších kroků mohlo být formální testování s nedostatkem paměti. To by mohlo dát rozsahem na celou další práci. Je nutné ověřit, že knihovna bude reagovat správným způsob a nezpůsobí pád systému v rámci kterého je spouštěna.

Literatura

[1] Aho, A. V.; Corasick, M. J.: Efficient String Matching: An Aid to Bibliographic Search. *Commun. ACM*, ročník 18, č. 6, Červen 1975: s. 333–340, ISSN 0001-0782, doi:10.1145/360825.360855.
URL http://doi.acm.org/10.1145/360825.360855

[2] Kim, K. S.; Sahni, S.: IP Lookup By Binary Search On Prefix Length. In *Proceedings of the Eighth IEEE International Symposium on Computers and Communications*, ISCC '03, Washington, DC, USA: IEEE Computer Society, 2003, ISBN 0-7695-1961-X, s. 77–.
URL http://dl.acm.org/citation.cfm?id=839294.843365

[3] Medhi, D.: *Network routing: algorithms, protocols, and architectures*. Morgan Kaufmann, 2010.

[4] Morris, J. M.: Traversing binary trees simply and cheaply. *Information Processing Letters*, ročník 9, č. 5, 1979: s. 197 – 200, ISSN 0020-0190, doi:http://dx.doi.org/10.1016/0020-0190(79)90068-1.
URL http://www.sciencedirect.com/science/article/pii/0020019079900681