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

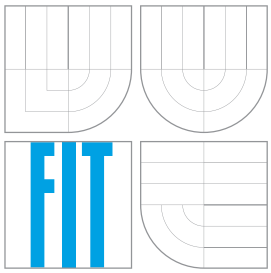
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

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 binárního vyhledávání na délce prefixu, 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

This thesis is focused on time-critical operations in context of computer networks. Processed operations are packet classification, specially one-dimensional classification, longest prefix matching using binary search on prefix length and TreeBitmap, pattern matching using Aho-Corasick, regular expression matching and packet header analysis and extraction. Purpose of this work is to design API for library implementing these operations. Implementation speed of these operations is measured on Intel and ARM platforms.

Klíčová slova

počítačové sítě, hledání nejdelšího shodného prefixu, hledání řetězců, regulární výrazy, binární vyhledávání na délce prefixu, TreeBitmap, Aho-Corasick

Keywords

computer network, longest prefix matching, pattern matching, regular expressions, binary search on prefix length, TreeBitmap, Aho-Corasick

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 Ing. Jana Kořenka, Ph.D.

.....
Lukáš Vokráčko
19. května 2015

Poděkování

Chtěl bych poděkovat vedoucímu této práce Ing. Janovi Kořenkovi, Ph.D. za jeho odborné vedení, cenné rady a poskytnutý čas.

Obsah

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. Rychlostí s jakou přibývají nový uživatelé internetu a zařízení vyžadující přístup k počítačovým sítím se neustále zvyšují požadavky na rychlost, se kterou data prochází počítačovými sítěmi a z toho vyplývající požadavky na rychlost zpracování síťového provozu. Největší požadavky jsou kladeny na zařízení starajících se o řízení internetového provozu na páteřních linkách. Mezi tyto zařízení lze zařadit směrovače, které řídí datové toky mezi jednotlivými sítěmi, přepínače 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 procházejícího sítí.

Páteřní spoje v době psaní této práce dosahují rychlostí v řádech desítek gigabitů za sekundu a z toho vyplývají požadavky na rychlost zpracování síťových dat. Nicméně je důležité, aby stejné rychlosti zpracování dosahovaly všechna zařízení na páteřních spojích, protože 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. Časově kritické operace jsou takové operace, jež při zpracování síťového provozu trvají nejdelší dobu, a rozbor takovýchto operací je součástí této práce. Z těchto operací jsou vybrány a rozvedeny operace hledání řetězců a regulárních výrazů, analýza a extrakce hlaviček paketů a klasifikace paketů, speciálně pak jednodimenzionální klasifikace dle cílové IP adresy, hledání nejdelšího shodného prefixu.

Přínosem této práce je implementace algoritmů provádějících zmíněné časově kritické operace v podobě knihovny, která bude využita výzkumnou skupinou ANT na Fakultě Informačních technologií Vysokého učení technického v Brně pro vytváření bezpečnostních systémů a aplikací.

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 této knihovny a možnosti rozšíření o další časově kritické operace. V kapitole 4 jsou vizualizovány a diskutovány výsledky, jichž se podařilo dosáhnout při implementaci zmíněných operací a to na dvou hlavních platformách,

[1] Intrusion detection system
[2] Intrusion prevention system

Intel a ARM. Kapitola 5 shrnuje dosažené výsledky a nastiňuje další možný vývoj této knihovny.

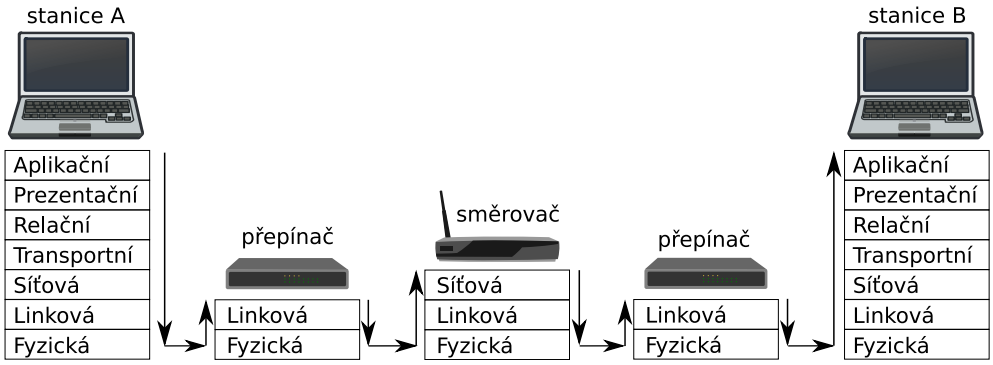
Kapitola 2

Teoretický rozbor

Tato kapitola poskytuje teoretické informace, jež tvoří základ této práce. V první části je popsán síťový model ISO/OSI a jeho vrstvy důležité z pohledu této práce. V druhé části jsou pak podrobněji rozebrány jednotlivé časově kritické operace a největší důraz je kladen na jednodimenzionální klasifikaci paketů, vyhledání nejdelšího shodného prefixu.

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 využití našel pouze v akademické sféře. V reálných počítačových sítích pak dominuje model TCP/IP, který má oproti ISO/OSI modelu menší počet vrstev. Model 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 vrstvy. Ty jsou detailněji popsány v následující části. Na obrázku 2.1 je znázorněn průchod jednoho datového paketu odeslaného ze stanice A na cílovou stanici B jednotlivými vrstvami modelu ISO/OSI. Jak je z obrázku patrné, tak různé druhy síťových zařízení pracují s různými vrstvami.



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

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

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 a 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, která je vysvětlena v následujícím odstavci. Síťová zařízení pracující na této vrstvě se nazývají přepínače (angl. *switch*). Úkolem přepínačů je odeslat vstupní data portem, který vede k cílovému zařízení.

Síťová vrstva se stará o adresaci zařízení připojených do internetu pomocí síťových adres a o směrování paketů. Nejrozšířenějším protokolem této vrstvy je protokol IP, jež existuje ve dvou verzích a to IPv4 a IPv6. Síťová vrstva umožňuje komunikovat zařízením, které nejsou spojeny přímo, ale existuje mezi nimi jedna nebo více cest napříč různými počítačovými sítěmi. Prvky pracující na této vrstvě jsou nazývány směrovače (angl. *router*) a pracují s datovou strukturou zvanou datagramy, jež obsahují právě IP adresy jednoznačně určující zdrojové a cílové zařízení. Směrování paketů je věnována kapitola 2.2.2.

Transportní vrstva umožňuje adresovat aplikace zodpovědné za přenášená data. Datová struktura této vrstvy je nazývána segment. Mezi dominující protokoly patří TCP a UDP. Hlavním rozdílem mezi těmito protokoly je zaručení spolehlivého doručení a vytváření trvanlivých spojení, které poskytuje pouze TCP. UDP naopak spolehlivé doručení negarantuje, ale díky tomu je tento protokol jednodušší. Situace, ve kterých pozitiva jako nižší režie přebijí zápory, je hlavně přenos dat v reálném čase. To je například streamování videa, přenos hlasu technologie VoIP nebo přenos informací do online her. Zpracování dat na úrovni transportní vrstvy a všech vyšších vrstev není implementováno na síťových zařízeních starajících se přenos dat po síti.

2.2 Časově kritické operace

Pod pojmem časově kritické operace se rozumí takové operace, které zabírají nejvíce výpočetního času při zpracování jednoho paketu a typicky je nutné provádět je na více síťových zařízeních. Těmito zařízeními mohou být směrovače, přepínače, firewally a také systémy oddělené od řízení síťového provozu jako například sondy 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 na jednodimenzionální klasifikaci dle cílové IP adresy, vyhledávání nejdelšího shodného prefixu. Tato operace je využívána pro prohledávání směrovací tabulky směrovačů pro určení nejvhodnější cesty, kterou bude paket pokračovat při své cestě k cílovému zařízení. Další z operací je analýza a extrakce hlaviček paketů, která je využívána v již zmíněné klasifikaci paketů, kde je nutné z hlavičky paketu extrahovat všechny informace, dle kterých bude paket klasifikován. Dalšími z rozebíraných operací je hledání řetězců a hledání regulárních výrazů. Poslední dvě zmíněné operace slouží především pro detekci útoků v systémech IDS (z angl. *intrusion detection system*) a pro prevenci útoků v systémech IPS (z angl. *intrusion prevention system*). Jedná se o operace sloužící pro hloubkové prohledávání paketů

(angl. *Deep Packet Inspection*). Toto prohledávání na rozdíl od ostatních operací pracuje s datovým obsahem paketů a ne jenom s hlavičkami paketu. Z toho vychází časová náročnost, neboť místo vyhodnocení hlavičky paketu, jež čítá 19 bytů pro IPv4 a 40 bytů pro IPv6 je nutno projít veškerá data, jejichž velikost se typicky pohybuje v rozmezí $1 - 1500B$.

Informace zde použité vycházejí z publikací [8], [9], [1], [11], [4] a [10].

2.2.1 Klasifikace paketů

Klasifikace paketů je operace rozhodující o dalším zpracování paketu. Výsledkem klasifikace pak může být rozhodnutí, zda daný paket může projít do dalšího vyhodnocování nebo zda pochází nebo směřuje do sítě, která není dovolena. Této klasifikace se využívá například pro povolení pouze určitého rozsahu zdrojových IP adres pro omezení přístupu do podnikové sítě nebo pro blokování paketů snažících se přistupovat ke službám, které jsou povoleny pouze pro specifická zařízení.

Data využívané pro klasifikaci se skládají z položek hlavičky paketu, pravidla a priority. Nejčastěji využívanou klasifikací je klasifikace skládající se z pětice položek IP hlavičky a to zdrojové adresy, cílové adresy, zdrojového portu, cílového portu a protokolu transportní vrstvy. Obsahem klasifikačních pravidel pak mohou být přesně specifikované hodnoty, rozsahy nebo prefixy. Prefixy jsou obecnějším zápisem specifických hodnot i rozsahů, neboť rozsah lze přepsat v nejhorším případě na $2N - 2$ prefixů [5], kde N odpovídá počtu bitů reprezentující rozsah. V případě specifické hodnoty je to prefix pouze jeden o bitové délce stejné jako reprezentovaná hodnota.

Klasifikace je prováděna jako vyhledání každé definované položky v množině reprezentující hodnoty této položky definované v klasifikátoru a poté výběr pravidla s nejvyšší prioritou z kartézského součinu množin obsahující vyhovující hodnoty jednotlivých položek.

Tato práce se zabývá pouze jednodimenzionální klasifikací paketů, jež je založena na klasifikaci dle cílové IP adresy, hledání nejdelšího shodného prefixu, operace sloužící pro prohledávání směrovacích tabulek směrovačů.

2.2.2 Hledání nejdelšího shodného prefixu

Problém hledání nejdelšího shodného prefixu se rozumí jednodimenzionální klasifikace paketů dle jejich cílové IP adresy, která může být jak verze 4, tak verze 6. Hledání nejdelšího shodného prefixu je operace, která je prováděna na síťových prvcích zvaných směrovače. Tyto prvky jsou umístěny na každém rozhraní dvou a více počítačových sítí. Cílem směrovačů je nalézt nejvhodnější cestu, kterou bude směrován příchozí paket. Struktura reprezentující uložené směrovací informace se nazývá směrovací tabulka. Tato tabulka ukládá informace o dostupných sítích (jejich prefixech), délce těchto prefixů a rozhraní, kterým se lze do odpovídající sítě dostat. Příklad směrovací tabulky je zobrazen v tabulce 2.1

Prefix	Délka prefixu	Rozhraní
147.228.0.0	14	eth0
147.228.128.0	17	eth0

Tabulka 2.1: Příklad směrovací tabulky

S velkým rozmachem počítačových sítí v poslední dekádě dochází k velkému nárůstu směrovacích informací a z toho vycházející nárůst velikosti směrovacích tabulek. Jako jedno

z řešení pro zmenšení směrovacích tabulek byl navržen takzvaný supernetting, který agreguje směrovací záznamy sdílející stejné rozhraní a mající společnou část prefixu do jednoho záznamu. Pokud vezmeme v úvahu výše uvedenou tabulku, tak při použití supernettingu by byly oba záznamy sloučeny v jeden, který by vypadal takto: 147.228.0.0/14 $eth0$. Tím je dosaženo zmenšení směrovacích tabulek, nicméně i s využitím supernettingu zůstává hledání nejdelšího shodného prefixu časově kritickou operací.

Prefix ve směrovací tabulce je interně reprezentován jako posloupnost nul a jedniček (binární vyjádření jeho hodnoty) s hvězdičkou na konci, která značí, že všechny adresy, jejichž začátek je shodný s částí před hvězdičkou, odpovídají tomuto prefixu. Jako příklad mějme prefix $A = 1001*$ jež odpovídá adresám začínajícím 1001, tedy $10010*$ i $10011*$. Nicméně ve směrovací tabulce může být uložen i prefix B $10010*$, který sdílí první čtyři bity své adresy s výše uvedeným prefixem A a v případě, že je zpracovávána adresa začínající hodnotou 10010 je z pohledu směrování nutno vyhodnotit prefix B jako nejdelší shodný 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 slouží pro rozhodnutí, jaké pravidlo směrovací tabulky odpovídá nejdelšímu shodnému prefixu. Délka prefixu může nabývat hodnot $1-32$ pro adresy typu IPv4 a $1-128$ pro adresy typu IPv6. Z výše uvedených informací a příkladů vyplývá, že je nutné rozlišovat, zda existující prefix reprezentuje adresu verze IPv4 nebo IPv6, jinak by mohlo docházet k vyhodnocení nejdelšího shodného prefixu pro adresu IPv4 jako prefix verze IPv6, což by mělo za následek nevalidní směrování paketů. Jako příklad může sloužit následující směrovací tabulka, ve které je první pravidlo verze IPv4 a druhé IPv6. Při vyhledání nejdelšího shodného prefixu v této tabulce by nebylo možné určit jaké pravidlo je to správné. Pro rozlišení o jaký druh prefixu se jedná je možno použít dva přístupy, rozlišení na úrovni záznamů směrovacích tabulek nebo rozlišením na úrovni směrovacích tabulek. V této práci je zvoleno rozlišení na úrovní směrovacích tabulek.

Prefix	Délka prefixu	Rozhraní
$100*$	3	eth0
$100*$	3	eth3

Tabulka 2.2: Příklad směrovací tabulky

Pro hledání nejdelšího shodného prefixu existuje velké množství algoritmů, které jsou popsány v [9]. Většina z nich je založena na procházení stromové struktury. Každý algoritmus má jiné paměťové nároky a dosahuje jiných rychlostí. Z toho důvodu je při výběru vhodného algoritmu nutné volit kompromis mezi rychlostí a paměťovou náročností. V případě, že jde o implementaci na architektuře FPGA, bude důraz pravděpodobně kladen na paměťovou náročnost a to z důvodu, že tyto čipy mají omezenou kapacitu paměti. Na architekturách vycházejících z x86 je naopak kladen důraz na rychlost zpracování z důvodu obecných procesorů, které nejsou specializovány na zpracování těchto operací a dosahují tak delší doby zpracování.

První algoritmus hledání nejdelšího shodného prefixu byl založen na naivním procházení lineárního seznamu. Doba vyhledávání pomocí tohoto algoritmu byla závislá na počtu uložených prefixů a její časová složitost byla $O(N)$, což při dnešních rychlostech spojů umožňuje procházet směrovací tabulku obsahující pouze malý počet záznamů.

Algoritmy rozebrané v této kapitole vycházejí z obecně vícebitového stromu, jež je rozšířením binárního stromu na více než dva potomky. Hodnota klíče uzlu v obecném vícebitovém stromu není přímo zanesena do uzlu jako jedna z jeho položek, ale odpovídá cestě

Prefix	∗	0∗	1∗	00∗	01∗	10∗	11∗
Interní bitmapa	0	1	0	0	0	1	1
Externí bitmapa				1	0	1	0

Tabulka 2.3: Příklad bitmap algoritmu TreeBitmap

stromem od kořene k aktuálnímu uzlu. Těmito algoritmy je binární vyhledávání na délce prefixu a TreeBitmap. Binární vyhledávání na délce prefixu používá binární strom pro interní reprezentaci směrovacích informací a pro vyhledávání využívá struktury hašovací tabulky, ve které jsou uloženy všechny existující uzly binárního stromu. TreeBitmap je algoritmus založený na vícebitovém stromu a vyznačuje se tím, že pro uložení prefixu využívá zakódované bitmapy a každý uzel stromu může uchovávat informace pro větší počet bitů prefixu, díky čemuž dosahuje menších paměťových nároků. Dosažené výsledky pro různý počet bitů jsou vizualizovány v kapitole 4.1.

TreeBitmap

Algoritmus TreeBitmap je založen na datové struktuře vícebitového stromu, v němž jsou uloženy směrovací informace. Hlavní myšlenkou tohoto algoritmu je uložení potomků uzlu a směrovacích pravidel na jednom paměťovém místě, což znamená, že stačí uchovávat pouze jednu adresu reprezentující paměťové místo, kde se uzly nacházejí, a index pro určení jaký uzel se má vybrat. Právě hodnota indexů je zakódovaná do bitmap. Výhodou tohoto zakódování informací je zmenšení paměťových nároků pro uložení každého uzlu stromové struktury. Další specifickou vlastností algoritmu TreeBitmap je zpracování několika bitů adresy v jednom kroku. Počet zpracovaných bitů je nazýván střída a právě velikost střídy určuje počet bitů prefixu zakódovaných do jednoho uzlu stromu.

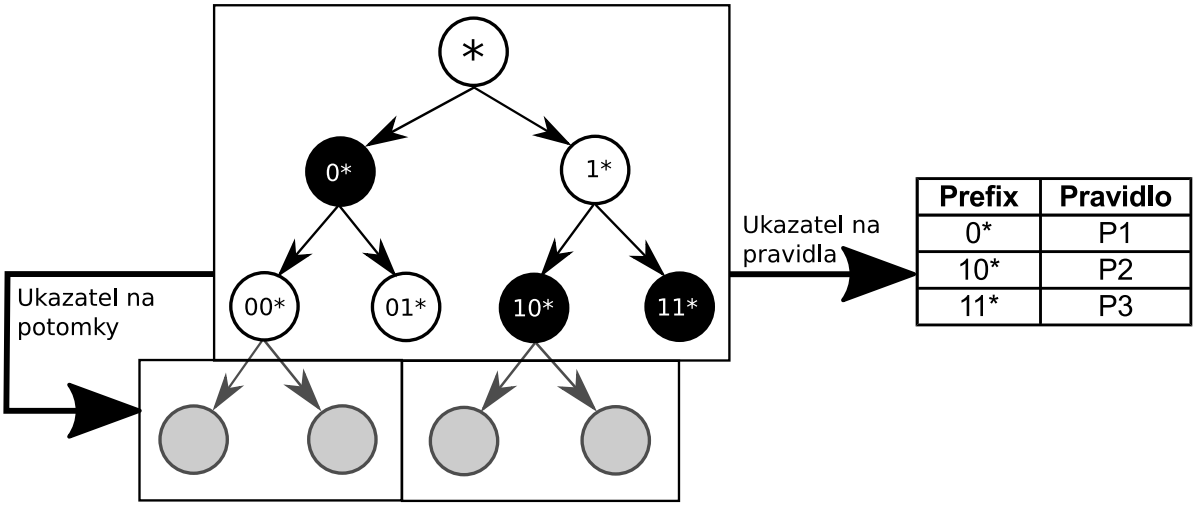
Každý uzel stromové struktury obsahuje dvě bitmapy, interní a externí, které slouží pro zakódování indexů. Interní bitmapa slouží pro zakódování informací, jaké prefixy a jim odpovídající pravidla se nacházejí v aktuálním uzlu. Externí bitmapa pak uchovává informace o existujících cestách do dalších úrovní stromové struktury. Příklad zakódovaných bitmap pro střídu 2 je zobrazen v tabulce 2.3. Další položkou uzlu je ukazatel na paměťové místo, kde jsou uloženy potomci uzlu, a ukazatel na pole obsahující směrovací pravidla, jež odpovídají adresám reprezentovaných tímto uzlem.

Algoritmus vyhledávání spočívá v procházení stromu od kořenu a zjišťování, zda existuje prefix odpovídající zpracované části adresy. Částí adresy se rozumí N bitů, jež jsou brány postupně od nejvýznamnějších bitů hledané adresy až po nejméně významné. Jako první operace hledání nejdelšího shodného prefixu se extrahují bity adresy na pozici odpovídající aktuální hloubce zanoření ve stromu a provede se operace zjištění, zda je pro část této adresy uloženo směrovací pravidlo. To je zjištěno z interní bitmapy na pozici, jež odpovídá délce a hodnotě extrahovaných bitů. Pozice v interní bitmapě je vypočítána jako $2^N - 1 + x$, kde N reprezentuje počet bitů (velikost střídy v první iteraci nad daným uzlem) a x je dekadickou reprezentací extrahovaných bitů. Pokud je na této pozici hodnota "1", je uloženo pravidlo na indexu spočítatelným jako `ones(bitmap, position)` v poli pravidel. Toto pravidlo pak reprezentuje dočasný nejlepší výsledek. Pokud je na vypočítané pozici v interní bitmapě hodnota "0" je opakován výpočet pozice s hodnotou N sníženou o jedna a bitovou hodnotou oříznutou o nejméně významný bit. Tato operace se opakuje tak dlouho, dokud není nalezena pozice s hodnotou "1"nebo dokud není N nulové. Jako druhá operace je provedeno hledání následovníků ve stromové struktuře reprezentující specifičtější prefixy.

To je provedeno jako zjištění přítomnosti hodnoty "1"v externí bitmapě na pozici, jejíž hodnota je dekadickou reprezentací extrahovaných bitů. V případě přítomnosti "1"na této pozici je proveden přechod do další úrovně stromové struktury a celý postup se opakuje. Pokud je na této pozici hodnota "0"je vyhledávání ukončeno a jako výsledek je navrácena hodnota dočasného nejlepšího výsledku, která odpovídá pravidlu, jež patří k nejdelšímu shodnému nalezenému prefixu. Celý algoritmus je popsán pseudokódem v algoritmu 2.1.

Pro výpočet indexu do polí obsahující pravidla a následovníky se používá funkce `ones(bitmap, position)`, jež spočítá počet bitů s hodnotou "1"v dané bitmapě a to od pozice 0 do pozice `position`.

Jeden uzel stromu TreeBitmap je vizualizován na obrázku 2.2 a jeho bitmapy odpovídají tabulce 2.3. Uzly zbarvené černě jsou uzly, pro které je definováno směrovací pravidlo.



Obrázek 2.2: Jeden uzel algoritmu TreeBitmap

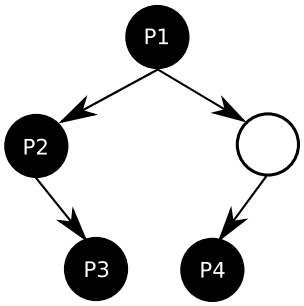
Algoritmus 2.1: Hledání nejdelšího shodného prefixu algoritmem TreeBitmap

```
1  node ← tbm-root;
2  longest-match-node ← tbm-root;
3  longest-match-index ← 0;
4  position ← 0;
5  repeat
6      bits ← get-stride-bits(ip, position);
7      position ← position + STRIDE;
8      if internal-index(node.internal, bits) then
9          longest-match-node = node;
10         longest-match-index = internal-index(node.internal, bits);
11     index ← ones(node.external, bits);
12     parent ← node;
13     node ← node.external[index];
14 until BIT(parent.external, bits);
15 return longest-match-node.rule[logest-match-index];
```

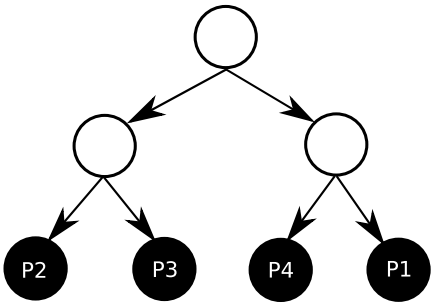
Binární vyhledávání na délce prefixu

Algoritmus binárního vyhledávání na délce prefixu vychází z binárního stromu, přidává operaci propagování listů (angl. *leaf-pushing*) a zavádí efektivnější prohledávání založené na hašovací tabulce. Struktura uzlu je rozšířena o položky prefix, délka prefixu a typ uzlu. Typy uzlu jsou dva a to interní uzel mající právě dva potomky a uzel reprezentující pravidlo, který nemá žádné potomky. Mimo binárního stromu používá tento algoritmus i hašovací tabulku, do které jsou zaneseny všechny uzly stromu a jako hodnota pro hašovací funkci je použita hodnota prefixu v daném uzlu. Tato struktura je poté využívána pro hledání nejdelšího shodného prefixu.

Operace propagace listů je operace zaručující, že existují právě dva následovníci uzlu (typ internal) nebo neexistuje žádný (typ prefix). Pokud dojde ke stavu, že existuje právě jeden následovník uzlu, je operací propagování uzlů vytvořen i druhý uzel a je do něj zaneseno pravidlo, jež obsahuje nadřazený uzel. Ukázku stromu před operací propagace listů je možno vidět na obrázku 2.3 a po provedení této operace na obrázku 2.4.



Obrázek 2.3: Strom před leaf-pushingem



Obrázek 2.4: Strom po provedení leaf-pushingu

Vyhledávání nejdelšího shodného prefixu v hašovací tabulce se skládá z následujících kroků. Jako první je provedeno hledání celé IP adresy, tedy všech 32 bitů v případě IPv4 nebo 128 bitů pro IPv6. Pokud je vyhledání úspěšné a nalezený prvek je typu prefix, pak je pravidlo tohoto uzlu navráceno jako nejdelší shodný prefix. V případě, kdy nalezený prvek je typu internal nebo není nalezen žádný prvek, dojde ke změně délky prefixu o hodnotu 2^{N-krok}, kde $N = 5$ pro IPv4 a $N = 7$ pro IPv6. V případě nalezení uzlu typu internal je délka adresy zvýšena o tuto hodnotu. Pokud není nalezen žádný prvek tak je délka adresy snížena o tuto hodnotu. Tento postup se opakuje, dokud není nalezen prvek typu prefix nebo je hodnota změny prefixu rovna nule. Postup hledání je zapsán pseudokódem v algoritmu 2.2. Příklad vyhledávání konkrétního prefixu ve směrovací tabulce 2.4 je popsán v tabulce 2.5.

Prefix	Délka prefixu	Pravidlo
147.228.0.0	14	P1
147.228.128.0	17	P2

Tabulka 2.4: Příklad směrovací tabulky

Operace vyhledání nejdelšího prefixu při využití binárního vyhledávání na délce prefixu má časovou složitost při ideální hašovací funkci $\log_2 N$, kde N je počet bitů adresy. Z principu algoritmu vyplývá, že nejhorší výsledky z časového hlediska bude dosahovat při shodě s prefixem, jehož délka je liché číslo. V takovém případě bude nutné projít všemi

Prefix	Délka	Uzel	Změna délky	Výsledek
147.228.128.54	32	Nenalezen	-16	
147.228.0.0	16	Interní	+8	
147.228.128.0	24	Nenalezen	-4	
147.228.128.0	20	Nenalezen	-2	
147.228.128.0	18	Nenalezen	-1	
147.228.128.0	17	Prefix	0	P2

Tabulka 2.5: Příklad vyhledání nejdelšího shodného prefixu

iteracemi vyhledávání. Počet kroků v případě IPv4 bude 5 a v případě IPv6 adresy to pak bude 7. Zde je vidět že i v případě 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.

Algoritmus 2.2: Hledání nejdelšího shodného prefixu s využitím binárního vyhledávání na délce prefixu

```
1  prefix-length ← ip-length;
2  prefix-change ← ip-length;
3  repeat
4  |  bits ← get-prefix-bits(ip, prefix-length);
5  |  item ← hash-table.get(bits);
6  |  prefix-change ← prefix-change ≫ 1;
7  |  if item == NULL then prefix-length ← prefix-length - prefix-change;
8  |  else if item.type == PREFIX then prefix-length ← prefix-length +
   |  prefix-change;
9  |  else break;
10 until prefix-change > 0;
11 if item == NULL then return bspl-root.default-rule;
12 return item.rule
```

2.2.3 Hledání řetězců

Jednou z častých operací při zpracování síťového provozu je hledání řetězců, jež je využíváno pro detekci signatur útoků na počítačové sítě, detekci malware a blokování dat obsahujících zakázaná klíčová slova. Hledání řetězců je ověřování, zda se jedno a více definovaných klíčových slov vyskytuje ve vstupních datech. V případě počítačových sítí se vstupními daty rozumí datových obsah paketů.

Pokud se oprostíme od počítačových sítí, tak dalším využitím hledání řetězců může být vyhledávání klíčových slov v textových dokumentech, což bylo podnětem pro vznik algoritmu autorů Aho a Corasickové, kteří tímto způsobem zrychlili prohledávání textových dokumentů až 5×. Alternativou k algoritmu Aho-Corasick může být považován algoritmus autorů Rabin–Karp [7], který má ovšem průměrnou časovou složitost $O(m+n)$, zatímco pro Aho-Corasick je tato složitost nejhorší možná. Algoritmem vycházejícím z Aho-Corasick a Boyer–Moore [2] je Commentz-Walter [3], jehož časová složitost však v nejhorším případě dosahuje $O(m * n)$. Algoritmem na nějž se soustředí část této práce je právě Aho-Corasick.

Tento algoritmus používá pro nalezení výskytu klíčového slova konceptu konečného automatu.

Konstrukce konečného automatu reprezentujícího klíčová slova je prováděna postupně a to tím způsobem, že je v automatu hledán již existující prefix vkládaného klíčového slova. Od výsledku tohoto hledání se pak vychází v dalších krocích, jež jsou následující:

Klíčové slovo $\leq$ Existující
: v tomto případě je pouze vloženo další pravidlo k uzlu reprezentujícímu poslední znak vkládaného klíčového slova

Klíčové slovo $>$ Existující
: v tomto případě je rozšířena již existující cesta a do posledního uzlu této cesty je přiřazeno odpovídající pravidlo.

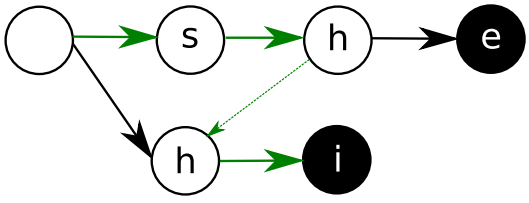
Po dokončení operace přidávání klíčových slov je provedeno generování tzv. *failure* přechodů. *Failure* přechod je mapováním přechodu, který je proveden v případě, že pro vstupní symbol neexistuje přechod z aktuálního stavu. *Failure* přechod pak reprezentuje prefix klíčových slov, které jsou podřetězcem aktuálně procházeného klíčového slova. Generování *failure* cesty je definováno iterativně a to následujícím způsobem: Pro počáteční uzel je *failure* cesta definována jako přechod do sebe sama. Pro každý uzel v úrovni 1 je *failure* cesta definována jako přechod do počátečního stavu. Pro každou další úroveň je možné *failure* přechod zjistit z validních přechodů stavů v nižších úrovních.

Příkladem procházení automatu a hledání klíčových slov může sloužit následující příklad reprezentovaný tabulkami 2.6 a 2.7 při vstupních datech `ship`.

Klíčové slovo	Pravidlo
she	1
hi	2

Tabulka 2.6: Klíčová slova

Automat reprezentující tyto klíčová slova je zobrazen na obrázku 2.5. Plnou čarou jsou znázorněny možné přechody, přerušovanou čarou *failure* přechody, černě jsou vybarveny stavy značící klíčové slovo a zelenou je znázorněn průchod automatem pro vstupní data `ship`.



Obrázek 2.5: Automat obsahující klíčová slova `she` a `hi`

Symbol [] reprezentuje počáteční stav a $[xyz]$ reprezentuje posloupnost stavů x, y a z, jež odpovídá cestě od počátečního stavu do aktuálního stavu.

Algoritmus procházení vstupních dat je popsán pseudokódem v 2.3 a vychází z [1].

Aktuální symbol	Akce automatu	Výsledek
s	[] $\xrightarrow{s}$ [s]	
h	[s] $\xrightarrow{h}$ [sh]	
i	$\nexists$ [sh] $\xrightarrow{i}$ [shi], [sh] $\xrightarrow{\epsilon}$ [h], [h] $\xrightarrow{i}$ [hi]	`hi`
p	$\nexists$ [hi] $\xrightarrow{p}$ [hip], [hi] $\xrightarrow{\epsilon}$ [], [] $\xrightarrow{p}$ []	

Tabulka 2.7: Prohledávání algoritmem Aho-Corasick

Algoritmus 2.3: Algoritmus procházení textu a hledání podřetězců

```
1 state = start-state;
2 for position ← 0 to text.length do
3 |   while (pos ← goto(state, text[position])) == FAIL do state ← state.failure;
4 |   if state.isMatch then return state.keyword;
5 |   state ← state.next[pos];
6 return NOT-MATCH;
```

2.2.4 Hledání regulárních výrazů

Hledání regulárních výrazů je, podobně jako hledání řetězců, operace sloužící pro detekci signatur síťových útoků a detekci škodlivého software v obsahu datových paketů v systémech IDS a IPS. Algoritmy pro hledání regulárních výrazů v rámci této práce jsou založeny na transformaci regulárních výrazů na konečné automaty. V této práci jsou rozebrány deterministické a nedeterministické konečné automaty.

Než se dostaneme k popisu jednotlivých druhů konečných automatů je potřeba definovat, co jsou to regulární výrazy a jaké operace je s nimi možno provádět.

Nechť r a s jsou regulární výrazy značící jazyky L_R a L_S. Regulární výrazy nad abecedou Σ a jazyky které značí, jsou definovány následovně:

- $\emptyset$ je RV značící prázdnou množinu (prázdný jazyk)
- ϵ je RV značící jazyk ϵ
- a, kde $a \in \Sigma$ je RV značící jazyk a
- $r \cdot s$ je RV značící jazyk $L = L_R L_S$
- $r + s$ je RV značící jazyk $L = L_S \cup L_S$
- $r*$ je RV značící jazyk $L = L_{R*}$

Nedeterministický konečný automat se od nedeterministického konečného automatu liší existencí ϵ-přechodů. $\mathcal{E}$-přechod umožňuje přejít do dalšího stavu konečného automatu bez nutnosti zpracovat vstupní symbol. Tvorba deterministického konečného automatu vychází z již existujícího nedeterministického konečného automatu a tento proces je zván determinizace. Determinizace odstraňuje ϵ-přechody pomocí operací *move* a *ϵ-closure*.

Při průchodu deterministický konečným automatem je možné zpracovat právě jeden znak vstupu v každé iteraci. Algoritmus projde všemi aktivními stavy a zjišťuje, zda existuje přechod pro aktuální znak na vstupu z aktuálního stavu a pokud existuje, tak je proveden přechod. V případě, že takový přechod neexistuje, tak je zpracovávaný stav označen jako

neaktivní a v dalším kroku již není zpracováván. V případě nedeterministického konečného automatu jsou navíc pro každý zpracovávaný stav aktivovány všechny stavy, kterých lze dosáhnout ϵ-přechodem. Algoritmus procházení deterministickým konečným automatem je popsán pseudokódem 2.4. Procházení nedeterministického automatu je popsáno pseudokódem 2.5.

Algoritmus 2.4: Procházení vstupních dat pro deterministický konečný automat

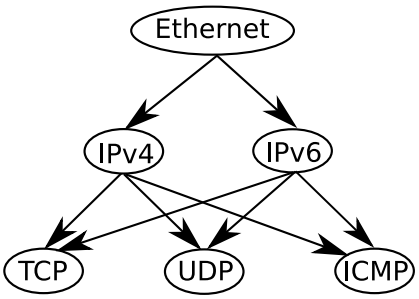
```
1 position ← 0;
2 while length > position do
3     symbol ← input[position];
4     states.push(root);
5     while state ← states.pop() do
6         if state.accepting then return state.rule;
7         if state.key[symbol] then states-new.push(state.next[symbol]);
8     swap(states, states-new);
9     position ← position + 1;
```

Algoritmus 2.5: Procházení vstupních dat pro nedeterministický konečný automat

```
1 position ← 0;
2 while length > position do
3     symbol ← input[position];
4     states.push(root);
5     while state ← states.pop() do
6         if state.accepting then return state.rule;
7         for i ← 0 to i < state.epsilon_count do states.push(state.epsilon[i]);
8         if state.key[symbol] then states-new.push(state.next[symbol]);
9     swap(states, states-new);
10    position ← position + 1;
```

2.2.5 Analýza a extrakce hlaviček paketů

Extrakce hlaviček paketů je operace prováděná na každém síťovém zařízení, neboť právě na základě hodnot položek hlavičky paketu je rozhodnuto, jak má být paket zpracován. Z toho vyplývá závislost rychlosti všech operací pracujících s položkami hlaviček na linkové, síťové a transportní vrstvě na rychlosti extrakce a analýzy hlaviček. Vstupními daty této operace jsou hlavičky datových struktur na vrstvě linkové, síťové a transportní. Výstupem je množina hodnot, jež byly cílem analýzy a extrakce. Parsování probíhá sekvenčně, neboť protokol dané vrstvy je uveden v hlavičce na vrstvě o úroveň níže. Operaci je možno rozdělit na dva kroky. Prvním krokem je lokalizace požadované hlavičky ve vstupních datech a druhým je pak extrakce této hodnoty. Jedním ze způsobů používaných pro lokalizace hlavičky paketů je použití orientovaných acyklických grafů zvaných parsovací grafy, kde vrcholy reprezentují typy hlaviček a hrany reprezentují posloupnost hlaviček. Příklad takového grafu lze najít na obrázku 2.6.



Obrázek 2.6: Parsovací graf

Kapitola 3

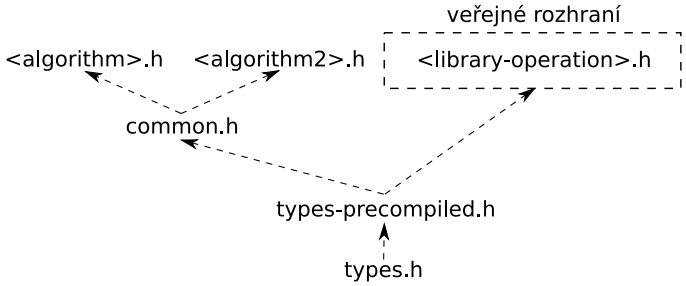
Návrh API knihovny

Cílem návrhu API knihovny `fastnet` je vhodně zapouzdřit implementované operace a to takovým způsobem, aby je bylo možné používat bez znalosti využívaných algoritmů. Dalším cílem návrhu je připravit API tak, aby bylo možné rozšiřovat množinu algoritmů implementující dané operace při zachování stejného API. Každá z funkcí navrženého API pro jednotlivé operace umožňuje pracovat s více datovými sadami, jež reprezentující data dané operace. To byl také jeden z požadavků při návrhu API. Kořenový prvek je explicitně předáván do funkcí provádějící jednotlivé operace namísto jeho uložení uvnitř knihovny. Jedinou výjimkou jsou inicializační funkce, které právě vytvářejí kořenovou strukturu.

Knihovna `fastnet` je rozdělena na několik 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 a klasifikaci paketů. Další výhodou tohoto rozdělení je možnost snadno vytvořit a používat jednotlivé knihovny samostatně nebo v různých kombinacích nezahrnující všechny operace. To se může hodit pro zařízení, která mají velmi limitované paměťové úložiště a jejich účelem je řešit pouze část ze zmíněných operací.

Veřejné rozhraní celé knihovny se skládá z veřejných rozhraní jednotlivých knihoven. Tyto rozhraní jsou popsány v následujících kapitolách. Pro knihovnu je navržena struktura hlavičkových souborů pro jednotlivé knihovny. Jako výchozí hlavičkový soubor je použit `types.h`, který obsahuje definice datových struktur pro všechny algoritmy v knihovně, které musí být viditelné i z veřejného rozhraní. Dalším souborem je `types-precompiled.h`, který je generován z `types.h`. Toto generování se provádí před samotným překladem knihovny a výsledek je závislí na zvoleném algoritmu. Soubor `common.h` je hlavičkový soubor společný pro všechny algoritmy v knihovně a `<algorithm>.h` pak obsahuje deklarace specifické pro právě jeden konkrétní algoritmus. Soubor `<library-operation>.h` je pak hlavičkový soubor, který tvoří veřejné rozhraní ke knihovním funkcím. Závislost jednotlivých souborů je znázorněna na obrázku 3.1.

Je navrženo API pro operace klasifikace paketů 3.1, hledání nejdelšího shodného prefixu 3.2, hledání řetězců 3.3, hledání regulárních výrazů 3.4 a analýzu a extrakci hlaviček paketů 3.5. V kapitole 3.6 je rozebrána možnost rozšíření této knihovny o další operace a v kapitole 3.7 je popsáno jakým způsobem je možné sestavit celou knihovnu nebo její jednotlivé části.



Obrázek 3.1: Diagram závislostí hlavičkových souborů

3.1 Klasifikace paketů

Pro klasifikaci paketů jsou navrženy čtyři základní funkce a jedna datová struktura. Tato struktura je nazvána `set` a obsahuje položky `rule` pro uložení pravidla, které odpovídá kombinaci ostatních položek, `dst` pro uložení cílové IP adresy, `src` pro uložení zdrojové adresy, `protocol` pro uložení typu transportního protokolu, tedy TCP nebo UDP a `port` pro uložení cílového portu. Výběr těchto položek vychází z nejčastějšího použití právě těchto položek a také z neexistující implementace této operace, která by vyžadovala definování zbylých položek.

Funkce pro práci se strukturou pro klasifikaci paketů jsou: `init` pro inicializaci vyhledávacích struktur, `add` pro přidání klasifikačního pravidla, `update` pro aktualizaci odpovídajícího pravidla, `remove` pro smazání odpovídajícího pravidla a nakonec `destroy` pro uvolnění veškeré paměti zabírané strukturami pro provádění klasifikace paketů.

Všechny zmíněné funkce a jedna datové struktura jsou navrženy ve dvou verzích a to pro IPv4 a IPv6. Tyto funkce se od sebe liší pouze prefixem, kde pro IPv4 je použit prefix `pc_` a pro IPv6 pak `pc6_`. Důvodem, proč jsou jednotlivé funkce a struktura rozděleny tímto způsobem, je odlišná velikost adres a nutnost rozlišovat pro jako verzi IP protokolu se klasifikace provádí, neboť prefixy adres se mohou shodovat pro obě verze IP.

Všechny funkce vyjma `init` očekávají jako první argument strukturu typu `root`, která obsahuje veškeré informace o klasifikačních pravidlech. Jako u ostatních operací je tato struktura uváděna explicitně a to umožňuje vytvářet více klasifikátorů v jednom programu.

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

3.2 Vyhledání nejdelšího shodného prefixu

Pro operaci vyhledání nejdelšího shodného prefixu jsou připraveny funkce `init`, `add`, `update`, `remove` a `destroy`. Dále je navržena struktura pro uložení kořene všech datových struktur, `root`. Všechny zmíněné funkce a datová struktura existují ve dvou verzích pro práci s jednotlivými verzemi protokolu IP. Odlišeny jsou prefixem, který je `lpm_` pro IPv4 a `lpm6_` pro IPv6.

Funkce `init` slouží pro vytvoření kořenového uzlu datových struktur a nastavení výchozího pravidla, které je vybráno, pokud není nalezen žádný odpovídající prefix. Návratovou hodnotou této funkce je ukazatel na kořen datové struktury, jež je parametrem všech ostatní funkcí provádějící operace v rámci vyhledání nejdelšího shodného prefixu. Funkce `add` slouží pro přidání prefixu a pravidla odpovídajícímu tomuto prefixu. Funkce `update` slouží pro aktualizaci pravidla daného prefixu. Funkce `remove` odstraňuje zvolený prefix z vyhledávací struktury. Funkce `destroy` slouží pro uvolnění paměti alokované všemi výše zmíněnými

funkcemi a po provedení této funkce již není možné provádět další operace nad danou strukturou.

Funkce pro vložení, smazání a aktualizaci 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.

Funkce pro vložení prefixu pracuje pouze s jedním prefixem a je prováděna okamžitě. Tento návrh vychází z předpokladu, že knihovna bude používána i v prostředí s dynamickými směrovacími protokoly jako například RIP, OSPF nebo BGP, které při změně směrovacích informací v případě BGP zasílají aktualizace s novými informacemi nebo jsou tyto změny zasílány periodicky v případě protokolu RIP.

Hašovací funkce je v této implementaci použita Jenkins [6]. Volba hašovací funkce může mít vliv na rychlost vyhledávání. To je patrné z principu hašovací tabulky, kde se kolizní záznamy uchovávají v lineárním seznamu, a v případě stejných hodnot generovaných hašovací funkcí pro všechny vstupní hodnoty by bylo vyhledávání omezeno na procházení lineárního seznamu.

Použitý algoritmus lze volit při sestavení knihovny pro vyhledání nejdelšího shodného prefixu zapsáním následujícího příkazu `make ALG=<alg>`, kde hodnota `<alg>` může nabývat hodnot `tbm` pro volbu TreeBitmap a `bspl` pro binární vyhledávání na délce prefixu. V případě sestavování celé knihovny je možné zapsat `make LPM=<alg>` v příslušném adresáři. Při volbě algoritmu TreeBitmap je pak volitelný parametr `STRIDE=<N>`, jež určuje velikost střídy. Tuto volbu je možné použít v obou příkazech.

Přesnou specifikaci rozhraní je možné nalézt v přiloženém CD ve složce `/lib/src/lpm/lpm.h` Specifikaci datových struktur lze nalézt v soubor `/lib/src/lpm/types.h`

3.3 Hledání řetězců

Pro hledání řetězců je navrženo několik datových struktur, mezi které patří struktura pro uložení právě jednoho stavu konečného automatu `_ac_state`, struktura reprezentující jeden konečný automat `pm_root` a struktura `pm_keyword` pro uložení klíčového slova, jeho délky a jemu odpovídající pravidlo.

Pro hledání řetězců jsou podobně jako pro vyhledání nejdelšího shodného prefixu navrženy a implementovány následující funkce.

- `pm_init` pro inicializaci datových struktur
- `pm_match` pro hledání prvního klíčové slova vyskytujícího se ve vstupních datech
- `pm_match_next` pro hledání dalších klíčových slov
- `pm_add` pro vložení množiny klíčových slov
- `pm_update` pro změnu pravidla odpovídající danému klíčovému slovu
- `pm_remove` pro smazání klíčového slova
- `pm_destroy` pro uvolnění veškeré paměti, jež byla alokována

Všechny výše zmíněné funkce vyjma `pm_init` a `pm_match_next` 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.

Hledání řetězců funkcí `pm_match` skončí svůj průchod konečný automatem v momentě nálezu první shody s libovolným klíčovým slovem zadaným při volání `pm_add`. V případě, že není nalezena žádná shoda s definovanými klíčovými slovy v řetězci, je vrácen výsledek `false`. Čtvrtým parametrem funkce `pm_match` může být hodnota `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 klíčovými slovy funkcí `pm_match_next`.

Jednotlivé položky struktury `pm_result` pro veřejné použití jsou

- `rule` - pole obsahující všechny nalezená klíčová slova
- `count` - počet nalezených klíčových slov
- `position` - pozice kde byl nalezen výskyt klíčového slova

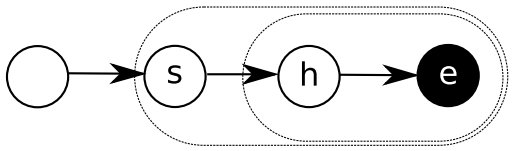
Pro práci se strukturou `pm_result` jsou v navrženy a implementovány následující operace:

- `pm_result_init` pro vytvoření této struktury
- `pm_result_destroy` pro uvolnění paměti alokované pro tuto strukturu

Funkce `pm_add` očekává jako druhý parametr pole struktur `pm_keyword`, kde každá struktura obsahuje položky vstupního klíčové slova, délku tohoto slova a pravidlo odpovídající tomuto slovu. Důvodem pro tento návrh je časově náročná funkce generování *failure* přechodů, které umožňují detekovat kratší klíčové slovo i v případě, že již je zahájeno porovnávání delšího slova, jak je možno vidět na následujícím příkladu. Tabulka 3.1 obsahuje definovaná klíčová slova a obrázek 3.2 znázorňuje část odpovídající konečného automatu pro klíčové slovo `she`, které v sobě obsahuje klíčové slovo `he`.

Klíčové slovo	Pravidlo
she	1
he	2

Tabulka 3.1: Klíčová slova



Obrázek 3.2: Konečný automat reprezentující 3.1

Přesnou specifikaci rozhraní je možné nalézt v přiloženém CD ve složce `/lib/src/pm/pm.h` Specifikaci datových struktur lze nalézt v soubor `/lib/src/pm/types.h`

3.4 Hledání regulárních výrazů

Rozhraní pro hledání regulárních výrazů navržené a implementované v knihovně `fastnet:regex` rozšiřuje množinu operací prováděných s regulárními výrazy zmíněných v 2.2.4 o následující zápisy operací:

- $[abc]$ 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 $1..N$ opakování
- $a?$ je definováno jako $0..1$ opakování
- . je definována jako libovolný symbol

Pokud se v definici regulárního výrazu vyskytuje jeden z řídících znaků regulárních výrazů je nutno tento znak escapovat, to znamená přidat před něj zpětné lomítko. Zpětné lomítko samotné se pak zapíše jako dvě zpětná lomítka. Operace konkatenace je uvažována implicitně.

Pro hledání regulárních výrazů jsou navrženy a implementovány dva způsoby hledání, deterministický a nedeterministický konečný automat. Z toho důvodu jsou odlišeny všechny funkce dle typu konečného automatu, který je použit. Pro použití deterministického konečného automatu je to `dfa` a pro nedeterministický je to `nfa`.

Navrženy jsou následující funkce:

- `regex_<type>_construct` pro vytvoření regulárního výrazu
- `regex_<type>_match` pro hledání regulárního výrazu ve vstupních datech
- `regex_<type>_destroy` a uvolnění paměti zabrané regulárním výrazem

Pro vytvoření regulárních výrazů je podobně jako u hledání řetězců použita pomocná struktura `regex_pattern`, která je předávána do funkce `regex_<type>_construct` a jež obsahuje pole regulárních výrazů, z nichž se má vytvořit jeden regulární výraz reprezentovaný konečným automatem. Výsledný konečný automat, ať už deterministický nebo nedeterministický, je výsledek spojení jednotlivých konečných automatů pro každý regulární výraz. Tím je umožněna detekce několika regulární výrazů v jednom průchodu vstupními daty i s přesnou identifikací jaký regulární výraz byl ve vstupních datech nalezen.

Důvodem proč jsou navrženy obě možnosti pro hledání regulárních výrazů je až teoreticky exponenciální nárůst počtu stavů deterministického konečného automatu, což může být nežádoucí a je vhodnější zaměnit rychlost deterministického konečného automatu za menší paměťové nároky nedeterministického konečného automatu.

Přesnou specifikaci rozhraní je možné nalézt v přiloženém CD v souboru `/lib/src/regex/regex.h` Specifikaci datových struktur lze nalézt v soubor `/lib/src/regex/types.h`

3.5 Analýza a extrakce hlaviček paketů

Pro tuto operaci byla navržena jedna funkce a jedna struktura. Touto strukturou je `phe_item` typu `union`, jehož položky se vzájemně překrývají. Důvodem pro tento návrh je předem neznámý počet extrahovaných hodnot a jejich typů. Navrženou funkcí je funkce `phe_get`, jež jako první parametr očekává vstupní data, jako druhý ukazatel na pole `phe_item`, kam se budou ukládat jednotlivé extrahované hodnoty, a další argumenty jsou hodnoty reprezentující jednotlivé položky hlaviček a jejich číselná hodnota odpovídá počtu bytů od začátku datové struktury paketu.

Některé základní položky využívané při klasifikaci paketů a jejich vzdálenosti od počátku hlavičky jsou uvedeny v tabulce 3.2.

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

Význam	Název	offset
Verze	IP_VERSION	0
Zdrojová IP adresa pro IPv4	IPv4_SRC	12
Cílová IP adresa pro IPv4	IPv4_DST	16
Zdrojový port TCP	TCP_SRC_PORT	0
Cílový port TCP	TCP_DST_PORT	2

Tabulka 3.2: Položky IP paketu a jejich pojmenování pro extrakci a analýzu

3.6 Rozšíření knihovny

Pro rozšíření knihovny je nutné přidat soubory implementující danou operaci do adresáře `lib/src/<operation>/` a upravit soubor `Makefile` v nadřazeném 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/<operation>/`. Další vhodnou součástí operace je benchmark pro vyhodnocení rychlosti jednotlivých implementací dané operace. Zdrojové soubory pro provádění benchmarků se ukládají do složky `lib/bench/<operation>/`.

3.7 Použití knihovny

Celou knihovnu je možné sestavit příkazem `make all` v hlavním adresáři knihovny. Při provedení tohoto příkazu bude celá knihovna sestavena bez debugovacích informací a s vypuštěním všech volání makra `assert`, které slouží pro kontrolu platnosti definovaných podmínek. Výsledkem sestavení je soubor knihovny `fastnet.a` nacházející se v adresáři `lib/src/`. Jako vedlejší produkt překladu vzniknou i knihovny implementující jednotlivé operace a budou umístěny v příslušných složkách `lib/src/<operation>/<operation>.a`. Dalšími cíli pro program `make` jsou:

- `test` - spustí automatické testování všech částí knihovny
- `bech` - spustí benchmarky všech částí knihovny
- `doc` - vygeneruje programovou dokumentaci ke všem částem knihovny
- `clean` - smaže všechny soubory vytvořené překladem

Knihovny reprezentující jednotlivé operace je možné sestavit příkazem `make` v odpovídajícím adresáři. Pro překlad s debugovacími informacemi je možno využít cíl sestavení `make lib`.

Kapitola 4

Výsledky

Tato kapitola shrnuje a vizualizuje dosažené výsledky při implementaci jednotlivých operací. Benchmarky byly provedeny na procesoru Intel(R) Core(TM) i3-2310M CPU @ 2.10GHz a pro architekturu ARM na desce SoCrates[1] s procesorem ARM Cortex-A9.

4.1 Hledání nejdelšího shodného prefixu

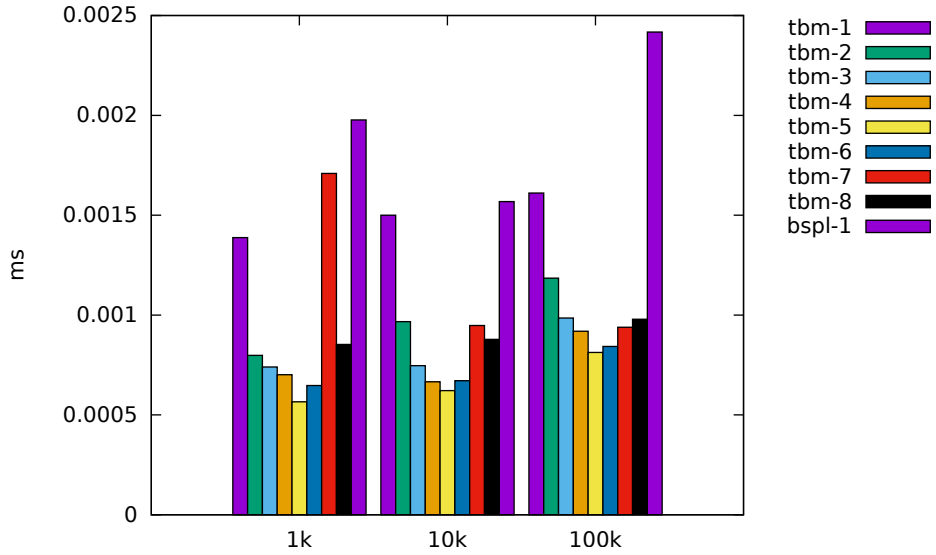
Hledání nejdelšího shodného prefixu bylo testováno na celkem pěti vstupních sadách dat, které se liší počtem záznamů směrovací tabulky. Data byla převzata z volně dostupných dat RIPE[2]. Počet záznamů směrovacích tabulek, verze IP adres a délky prefixů jsou popsány v tabulce 4.2. Dosažené výsledky pro jednotlivé sady dat jsou vizualizovány v grafech 4.1 pro sady IPv4 a 4.3 pro sady IPv6 na platformě Intel a v grafech 4.2 a 4.4 pro platformu ARM.

Název	Počet adres	Nejkratší prefix	Nejdelší prefix	Verze IP
1k	1000	13	31	IPv4
10k	10000	8	32	IPv4
100k	100000	8	32	IPv4
1k	1000	23	128	IPv6
10k	10000	19	128	IPv6

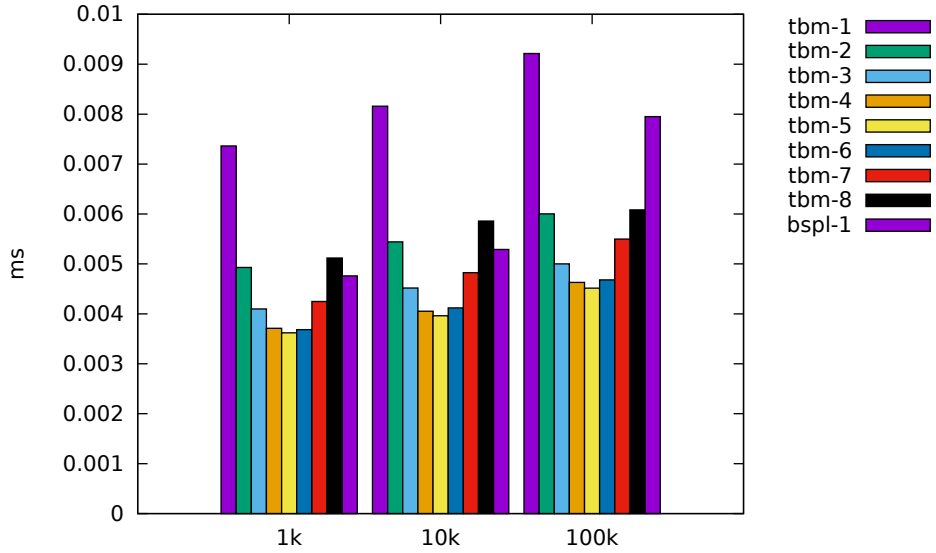
Tabulka 4.1: Směrovací tabulky pro testování

Vyhledávání ve směrovací tabulce bylo prováděno oproti 1000 IP adresám odpovídající verze. Tyto adresy byly vybrány ze záznamů směrovacích tabulek použitých pro testování.

[1] http://www.devboards.de/en/home/boards/product-details/article/socrates/
[2] https://www.ripe.net/



Obrázek 4.1: Rychlost hledání nejdelšího shodného prefixu pro IPv4 na platformě Intel

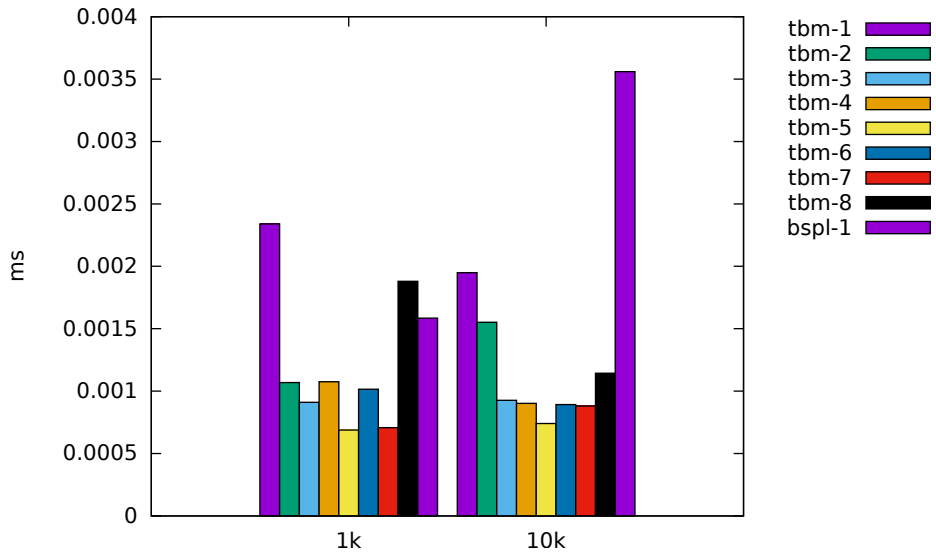


Obrázek 4.2: Rychlost hledání nejdelšího shodného prefixu pro IPv4 na platformě ARM

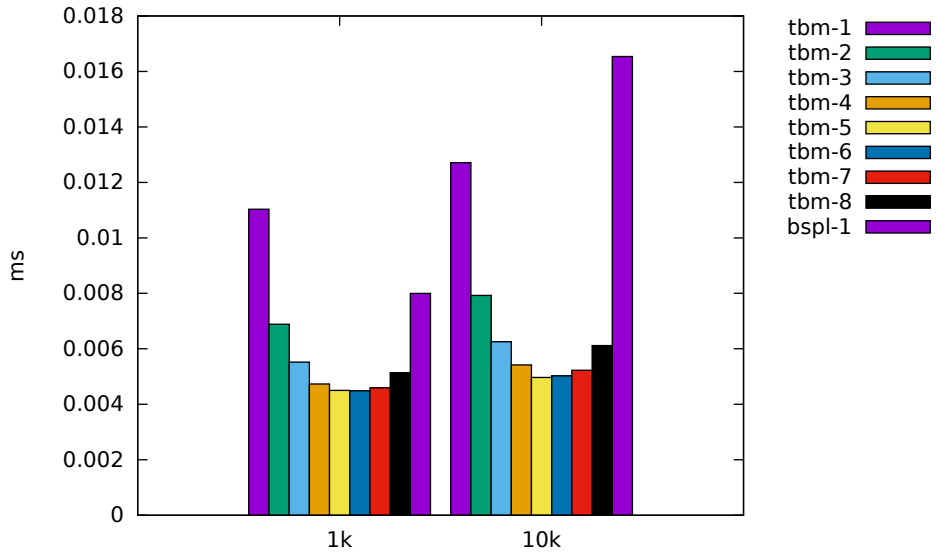
Benchmark byl proveden pro algoritmy TreeBitmap (v grafu označeno jako `tbm-střída`) ve velikostech střídy $1-8$ a pro binární vyhledávání na délce prefixu (v grafu označeno jako `bspl-1`). Jak je možno vyčíst z uvedených grafů, tak nejlepších výsledků bylo dosaženo pro TreeBitmap s velikostí střídy nastavenou na 5 bitů a to jak pro IPv4 tak i pro IPv6.

Pro testování algoritmu binárního vyhledávání na délce prefixu byla zvolena velikost hašovací tabulky stejná jako počet záznamů směrovací tabulky pro zvolenou sadu dat. Tento přístup byl zvolen z důvodu významného vlivu velikosti hašovací tabulky na rychlost samotného vyhledávání. V případě nastavení velikosti hašovací tabulky na 100 prvků byl algoritmus vyhledávání v tabulce obsahující 100000 záznamů 400× pomalejší.

Dalším důležitým faktorem pro efektivnost implementace je velikost datových struktur, která je závislá na velikosti střídy pro algoritmus TreeBitmap a také na počtu směrovacích



Obrázek 4.3: Rychlost hledání nejdelšího shodného prefixu pro IPv6 na platformě Intel



Obrázek 4.4: Rychlost hledání nejdelšího shodného prefixu pro IPv6 na platformě ARM

pravidel. Velikost struktury jednoho uzlu v závislosti na počtu směrovacích pravidel je ovlivněna následujícím způsobem. Do 256 záznamů zabírá pravidlo pouze $1B$, do 65536 záznamů pak $2B$ a pro více než 65536 pak celé $4B$ pro uložení právě jednoho pravidla. Rozdíl mezi uložení 65536 adres a 65537 adres pak bude činit $200KB$. Velikost uzlů stromu pro zvolené algoritmy v závislosti na počtu záznamů směrovací tabulky je popsán v tabulce 4.2.

		Velikost pro počet směrovacích pravidel		
Algoritmus	Velikost střídy	< 256	< 65536	≥ 65536
TBM	1	$24B$	$25B$	$28B$
TBM	2	$24B$	$25B$	$28B$
TBM	3	$24B$	$25B$	$28B$
TBM	4	$24B$	$25B$	$28B$
TBM	5	$32B$	$33B$	$36B$
TBM	6	$40B$	$41B$	$44B$
TBM	7	$64B$	$65B$	$68B$
TBM	8	$112B$	$113B$	$116B$
BSPL	—	$48B$	$49B$	$52B$

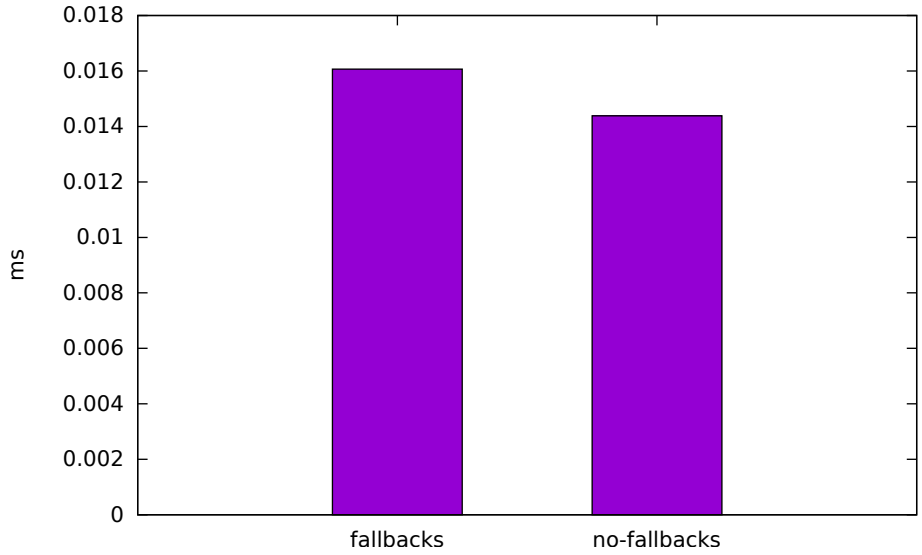
Tabulka 4.2: Velikosti datových struktur v závislosti na počtu směrovacích pravidel a velikosti střídy

Jak je možné vyčíst z výše uvedené tabulky, tak i přestože nejrychlejší implementací je TreeBitmap se střídou 5, tak v případě omezené paměti by bylo vhodnější zvolit kompromis mezi rychlostí a paměťovou náročností v podobě TreeBitmap s velikostí střídy 4, který dosahuje nejmenší možné velikosti datové struktury a zároveň nejlepších výsledků pro danou velikost datové struktury.

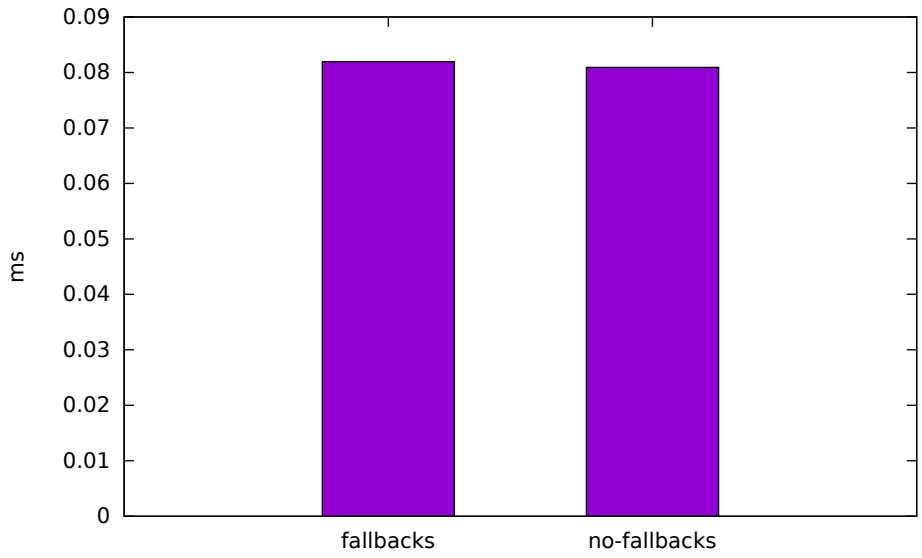
4.2 Hledání řetězců

Testování efektivity implementace hledání řetězců bylo provedeno na dvou vstupních testovacích sadách, kde první sada `fallbacks` obsahuje slova s výskytem stejných písmen, tudíž při tvorbě konečného automatu bude docházet ke generování *failure* přechodů, jež budou poté procházeny. Druhá testovací sada `no-fallbacks` obsahuje slova, která neobsahují stejná písmena. Výsledky dosažené pro jednotlivé testovací sady jsou vizualizovány v grafu 4.5 pro platformu Intel a 4.6 pro platformu ARM. Testování probíhalo na datových paketech odchycených z komunikace osobního počítače. Testovací sady obsahují klíčová slova používaná v protokolu HTTP[3].

[3]Hyper-text transfer protocol



Obrázek 4.5: Rychlost hledání řetězců na platformě Intel



Obrázek 4.6: Rychlost hledání řetězců na platformě ARM

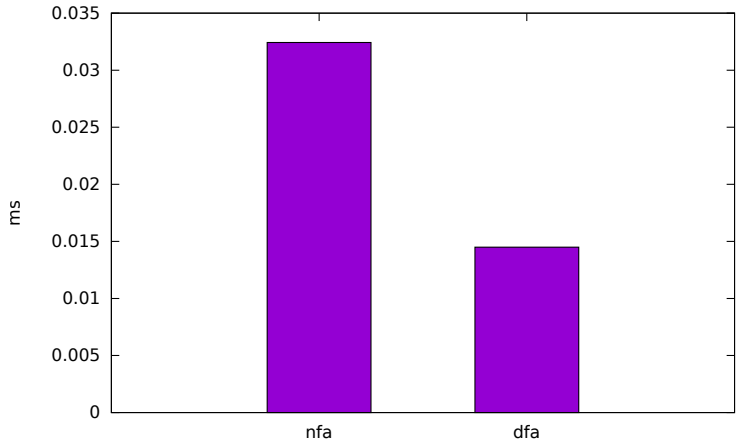
Jak je možné vyčíst z grafu tak sada obsahující *failure* přechody dosahuje mírně horších výsledků, což je způsobeno delším zpracování znaku, pro který neexistuje validní přechod v aktuálním stavu konečného automatu.

Velikost každého stavu konečného automatu je v této implementaci $48B$ a podobně jako hledání nejdelšího shodného prefixu je závislá na počtu vložených pravidel.

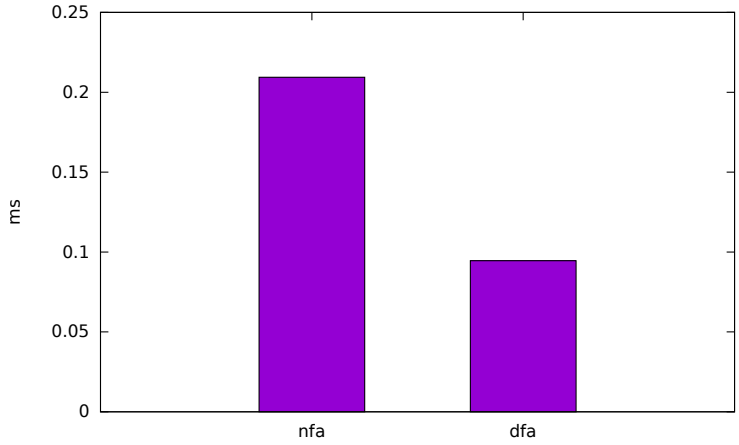
4.3 Hledání regulárních výrazů

Jak je vidět na v grafech 4.7 a 4.8 tak hledání regulárních výrazů pomocí deterministického konečného automatu je rychlejší více než dvakrát. Tento rozdíl v rychlosti je způsobený

nutností procházení ϵ-přechodů. Procházení založené na deterministickém konečném automatu tímto problémem netrpí, neboť při zpracování každého vstupního symbolu dojde k přechodu do dalšího stavu. Benchmark byl prováděn na datech reprezentující síťovou komunikaci osobního počítače a jako regulární výrazy byly použity výrazy definující URL adresu a stavové kódy HTTP protokolu.



Obrázek 4.7: Rychlost hledání regulárních výrazů na platformě Intel



Obrázek 4.8: Rychlost hledání regulárních výrazů na platformě ARM

Velikost datových struktur pro jednotlivé stavy konečného automatu pro deterministický a nedeterministický automat je znázorněna v tabulce 4.3.

Typ automatu	Velikost stavu
deterministický	$24B$
nedeterministický	$40B$

Tabulka 4.3: Velikosti stavů konečného automatu

Kapitola 5

Závěr

Cílem této práce bylo popsat a navrhnout aplikační programové rozhraní časově kritických operací využívaných v oblasti počítačových sítí. Teoretické základy, ze kterých tato práce vychází, jsou rozvedeny v kapitole 2. Mezi vybrané časově kritické operace patří klasifikace paketů a speciálně pak jednodimenzionální klasifikace paketů dle cílové IP adresy, hledání nejdelšího shodného prefixu. Pro tuto operaci jsou popsány algoritmy binárního vyhledávání na délce prefixu a TreeBitmap. Mezi další rozvedené operace patří hledání řetězců a pro tuto operaci je popsán algoritmus autorů Aho a Corasickové. Další z operací je hledání regulárních výrazů za použití konečných automatů, konkrétně deterministického a nedeterministického. Poslední popsanou operací je analýza a extrakce hlaviček. Pro každou popsanou operaci je navrženo API popsané v kapitole 3.

V kapitole 4 jsou diskutovány výsledky dosažené při implementaci operací hledání nejdelšího shodného prefixu, hledání řetězců a hledání regulárních výrazů. Pro hledání nejdelšího shodného prefixu vychází jako nejrychlejší implementace algoritmu TreeBitmap s velikostí střídy 5. Tohoto výsledku bylo dosaženo na několika datových sadách, jež vycházejí ze směrovacích tabulek reálný směrovačů. Pro operaci hledání řetězců bylo experimentování prováděno na síťovém provozu zachyceném při komunikaci osobního počítače. Jako vzorek testovaných klíčových slov bylo využito klíčových slov definovaných pro HTTP. Při experimentování s regulárními výrazy byly jako vstupní data použita stejná data jako pro hledání řetězců, a jako regulární výrazy byly použity takové regulární výrazy, které dokáží identifikovat URL[1] adresu a stavový kód HTTP protokolu. Pro regulární výrazy bylo dosaženo více než dvojnásobné rychlosti zpracování při využití deterministických konečných automatů oproti použití nedeterministických konečných automatů.

Jako kroky navazující na tuto práci je možné implementovat zbývající operace, pro které je navrženo API, ale nebyla provedena implementace. Těmito operacemi je klasifikace paketů a analýza a extrakce hlaviček paketů.

Ze specifikace požadavků na implementaci knihovny vychází požadavek na běh knihovny v prostředí vestavěných systémů, které disponují omezeným operační paměti, a z toho důvodu by jedním z dalších kroků mohlo být testování s omezenou velikostí operační paměti. Je nutné ověřit, že knihovna bude reagovat správným způsobem na nedostatek paměti a nezpůsobí pád systému, v rámci kterého je spouštěna.

Jako dalším možným rozšířením je návrh API a poté implementace vláknového zpracování, které bude umožňovat zřetězené zpracování jednotlivých operací.

[1] Uniform resource locator

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] Boyer, R. S.; Moore, J. S.: A Fast String Searching Algorithm. *Commun. ACM*, ročník 20, č. 10, Říjen 1977: s. 762–772, ISSN 0001-0782, doi:10.1145/359842.359859.
URL http://doi.acm.org/10.1145/359842.359859

[3] Commentz-Walter, B.: A String Matching Algorithm Fast on the Average. In *Proceedings of the 6th Colloquium, on Automata, Languages and Programming*, London, UK, UK: Springer-Verlag, 1979, ISBN 3-540-09510-1, s. 118–132.
URL http://dl.acm.org/citation.cfm?id=646233.682242

[4] Gibb, G.; Varghese, G.; Horowitz, M.; aj.: Design principles for packet parsers. In *Architectures for Networking and Communications Systems (ANCS), 2013 ACM/IEEE Symposium on*, Oct 2013, s. 13–24, doi:10.1109/ANCS.2013.6665172.

[5] Gupta, P.: *Algorithms for routing lookups and packet classification*. Dizertační práce, Stanford University, 2000.

[6] Jenkins, B.: A hash function for hash Table lookup. 2006.
URL www.burtleburtle.net/bob/hash/doobs.html

[7] Karp, R. M.; Rabin, M.: Efficient randomized pattern-matching algorithms. *IBM Journal of Research and Development*, ročník 31, č. 2, March 1987: s. 249–260, ISSN 0018-8646, doi:10.1147/rd.312.0249.

[8] 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

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

[10] Meduna, A.: *Automata and languages: theory and applications*. Springer Science & Business Media, 2000.

[11] Partridge, C.: *Gigabit Networking*. Addison-Wesley professional computing series, Addison-Wesley, 1994, ISBN 9780201563337.
URL http://books.google.com.au/books?id=GUZGHzL-bM4C

Příloha A

Obsah CD

- `/lib/src/` zdrojové kódy knihovny
- `/lib/bench/` benchmarky pro jednotlivé operace
- `/lib/test/` testovací skripty
- `/lib/doc/` programová dokumentace
- `/text/` zdrojové kódy textu práce
- `xvokra00.pdf` tato zpráva