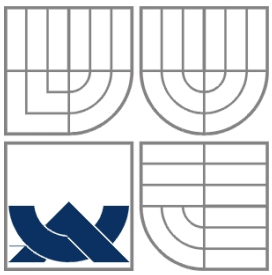
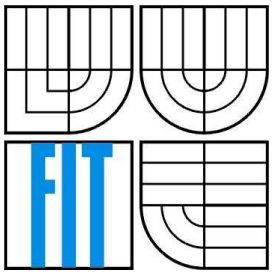


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KONSTRUKCE NEDETERMINISTICKÝCH KONEČNÝCH AUTOMATŮ

CONSTRUCTION OF NONDETERMINISTIC FINITE AUTOMATA

BAKALÁŘSKÁ PRÁCE
BACHELOR´S THESIS

AUTOR PRÁCE — TIMOTEJ STANEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2009

Abstrakt

Tato práce pojednává o problematice konstrukce nedeterministických konečných automatů z PCRE regulárních výrazů s ohledem na jejich parametry při použití v IDS systémech. Rozebraná je také gramatika PCRE výrazů. Dále jsou uvedeny dva rozdílné přístupy ke konstrukci nedeterministických konečných automatů z PCRE výrazů. Následně je popsána implementace jednotlivých algoritmů. Pomocí nich jsme sestrojili konečné automaty z výrazů z IDS systémů SNORT, Bro IDS a L7-Filter, a výsledné parametry porovnali a vyvodili závěry.

Klíčová slova

PCRE výrazy, regulární výrazy, nedeterministické konečné automaty, systémy detekce útoků, gluškovův konečný automat, thompsonův konečný automat, Bro IDS, L7-Filter, SNORT, hledání vzorů

Abstract

This thesis discuss about dilemma in construction of nondeterministic finite automata from PCRE expressions with respect of their parameters with use in Intrusion Detection Systems. There is showed PCRE expressions syntax too. We discussed two different approaches to construct nondeterministic finite automata from PCRE expressions. The implementation of these two algorithms is described. We constructed finite automata with them from expressions of three Intrusion Detection Systems: SNORT, Bro IDS and L7-Filter, and finally we compared their parameters and deduced conclusions.

Keywords

PCRE expressions, regular expressions, nondetermistic finite automata, intrusion detection system, Glushkov finite automata, Thompson finite automata, Bro IDS, L7-Filter, SNORT, pattern matching

Citace

Stanek Timotej: Konštrukcia nedeterministických konečných automatov v IDS systémoch. Brno, 2009, bakalářská práce, FIT VUT v Brně.

Konstrukce nedeterministických konečných automatů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Kaštila. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

……
Timotej Stanek
20. května 2009

Poděkování

Rád bych poděkoval vedoucímu práce Ing. Janovi Kaštilovi za poskytnuté materiály a projevenou snahu pomáhat.

Obsah

1 Úvod

Žijeme v informačnej dobe. Jednoduchý prístup k internetu je možný takmer pre každého človeka. Veľmi radi využívame jednoduchosť získania informácií a tiež poskytnutia informácií iným ľudom do celého sveta. Získavame a zdieľame informácie verejné ale taktiež aj súkromné. Súkromné informácie by mali byť prístupné iba subjektom, ktoré sú na to oprávnené.

Bohužiaľ existujú ľudia, ktorí zámerne kradnú súkromné informácie pre svoj prospech. Veľká časť z nich sa dá slušne speňažiť, či už sú to heslá k rôznym službám alebo bankovým účtom. Preto nastáva potreba sa proti takýmto praktikám chrániť.

Veľa informácií sa dnes prenáša po počítačových sieťach, práve preto je najčastejší terč internetových kriminálnikov. Prvoplánovou ochranou môže byť firewall alebo antivírus. Ďalším typom ochrany môžu byť tzv. Intrusion Detection Systems. IDS je systém, ktorý dokáže sledovať počítačovú sieť a hľadať v nej informácie o možných narušeniach siete. Dokáže detekovať viacero typov narušení, ktoré sa dajú vyjadriť pomocou PCRE výrazov.

PCRE výrazy využívajú IDS systémy ako Netfilter, Snort alebo Bro. Tieto výrazy dokážeme previesť na konečné automaty a následne pomocou nich hľadať vzory v sieťovej komunikácii. Tieto aplikácie sú určené pre klasické procesory, avšak spracovanie konečných automatov by mohlo byť rýchlejšie pomocou paralelizácie. Vďaka tomu by sme mohli sledovať väčšie dátové toky a viacej typov útokov naraz.

Paralelizovať spracovanie nám môžu pomôcť FPGA čipy. Preto sa hľadá vhodný spôsob ako previesť regulárne výrazy na konečné automaty s takými vlastnosťami, aby mohli byť vhodne použité v týchto čipoch. Tu sa nám ponúka riešenie použiť nedeterministické konečné automaty, ktoré by dokázali zrýchliť hľadanie vzorov.

Popíšeme si, čo sú PCRE výrazy a k čomu slúžia a vypíšeme si ich gramatické konštrukcie. Pozrieme sa ako fungujú konečné automaty a aké sú ich základné vlastnosti. Rozoberieme si postupy konštrukcie nedeterministických konečných automatov. Súčasťou práce je vytvorenie programu, ktorý dokáže previesť regulárne výrazy na nedeterministické konečné automaty a vytvoriť z nich štatistiky o ich vlastnostiach. Zo štatistík sme vytvorili grafy a mohli sme porovnať jednotlivé typy konštrukčných algoritmov.

2 PCRE výrazy

Perl compatible regular expressions (PCRE) je súbor pravidiel pre regulárne výrazy kompatibilné s regulárnymi výrazmi používanými v programovacom jazyku Perl. PCRE neslúži iba na vyhľadávanie vzorov v sieťovej komunikácii.

PCRE je knižnica, ktorá má vlastné aplikačné rozhranie, ale taktiež disponuje funkciami, ktoré sú určené pre aplikačné rozhranie normy POSIX. Táto knižnica, je napísaná v jazyku C, takže jej implementácia je prenositeľná medzi rôznymi platformami.

PCRE výrazy sú používané na definovanie sieťových útokov v IDS systémoch. Sú využívané napríklad na definovanie L7 pravidiel alebo programom Snort. Bližšie si rozoberieme jednotlivé gramatické pravidlá a na príkladoch si ukážeme ich použitie a konečné automaty, ktoré sa z nich vytvárajú.

Nie všetky PCRE konštrukcie sa dajú previesť na konečný automat. Taktiež nie všetky konštrukcie sa dajú previesť na konečný automat, bez toho aby sme neporušili niektoré pravidlo pre ich konkrétnu konštrukciu. Napríklad Gluškovov prístup nedokáže spracovať kvantifikátory {m,n}. Niektoré tieto konštrukcie sme sa rozhodli v implementácii vynechať. Konkrétne sme do tejto skupiny zahrnuli spätné referencie, ktoré si rozoberieme v kapitole 4.1.

2.1 Pravidlá

V nasledujúcej podkapitole si ukážeme aké gramatické pravidlá PCRE definuje a do akých tried ich rozdeľuje [1].

PCRE rozlišuje jednotlivé pravidlá na:

- Metaznaky
- Kvantifikátory
- Escape sekvencie
- Triedy znakov (character class)

Metaznaky sú znaky určujúce základné operácie. Je to znak, ktorý nesie špeciálny význam. Môže určovať operáciu alebo nejakým spôsobom meniť význam iných znakov. Do tejto kategórie patria:

\	*znak nasledujúci za \ nebude braný ako metaznak, ale ako obyčajný znak, ak sa nejedná o escape sekvenciu*
^	*vyjadruje znak začiatku riadku*
.	*označuje akýkoľvek znak, okrem nového riadku*

`$`	*značí koniec riadku*
`|`	*operátor alternácie*
`(? )`	slúžia *pre vyjadrenie rôznych konštrukcií, napr. pre spätné referencie* [4]
`( )`	*zátvorky upravujú prednosť operátorov*
`[]`	*pomocou hranatých zátvoriek definujeme triedy znakov*

Kvantifikátory určujú početnosť výrazu, na ktorý sa kvantifikátor viaže. Za každým kvantifikátorom môže byť jeden zo znakov ? alebo +, ktoré nám určujú aký dlhý reťazec máme hľadať [3].

`+`	*minimálne 1 výskyt*
`*`	*minimálne 0 výskytov*
`?`	*0 alebo 1 výskyt*
`{m}`	*určuje presne **m** výskytov*
`{m,}`	*minimálne **m** výskytov*
`{m,n}`	*minimálne **m** výskytov, maximálne **n** výskytov, musí platiť* $m < n$

Escape sekvencie združujú znaky, ktoré je potrebné zapísať špeciálnym zápisom. Na zápis sa používa už spomínaný metaznak „\“. Escape sekvencie využívajú taktiež triedy znakov. Patria sem rôzne znaky. Najčastejšie využívané v regulárnych výrazoch pre IDS systémy sú:

`\t`	*tabulátor*
`\n`	*nový riadok*
`\x54`	*znak zapísaný hexa hodnotou*
`\054`	*znak zapísaný oktalovou hodnotou*

Triedy znakov využívajú na zápis dvojicu metaznakov **[]** pre zápis podľa normy POSIX. Pomocou konkrétnej triedy vyjadrujeme možnosť dosadenia znaku, ktoré trieda obsahuje. Zapisujeme ich nasledujúcimi spôsobmi:

`[abc]`	*trieda obsahuje znaky a, b, c*
`[^abc]`	*trieda obsahuje všetky znaky okrem znakov a, b, c*
`[x-y]`	*definuje znaky s ASCII hodnotou v rozsahu* $\langle x, y \rangle$
`[[:class:]]`	*trieda obsahuje znaky z POSIXovej sady znakov „class“*
`[[^:class:]]`	*trieda obsahuje všetky znaky okrem znakov POSIXovej sady „class“*

Sady znakov definované v norme POSIX zhromažďujú znaky, ktoré sú logicky prepojené a uľahčujú nám písanie regulárnych výrazov. Vymenujeme si všetky sady, ktoré môžeme použiť pri PCRE výrazoch [2]:

`[[:alnum:]]`	*alfanumerické znaky*
`[[:alpha:]]`	*abecedné znaky*
`[[:ascii:]]`	*znaky s číselnou hodnotou 0-127 v ASCII tabuľke*
`[[:blank:]]`	*medzera, tabulátor*
`[[:digit:]]`	*desiatkové číslice*
`[[:xdigit:]]`	*hexadecimálne číslice*
`[[:space:]]`	*biele znaky*
`[[:upper:]]`	*veľké písmená*
`[[:lower:]]`	*malé písmená*
`[[:print:]]`	*špeciálne znaky z ascii vyjadrujúce*
`[[:word:]]`	*neznačí slovo, ale iba 1 znak - alfanumerický znak alebo „_“*
`[[:punct:]]`	*interpunkčné znaky*

2.2 Derivačný strom regulárneho výrazu

Derivačný strom je graf, ktorý reprezentuje regulárny výraz. Táto reprezentácia v sebe zahŕňa prednosť jednotlivých pravidiel pred inými pravidlami.

Z derivačného stromu môžeme jednoduchšie konštruovať konečné automaty, pretože nemusíme brať do úvahy prednosti jednotlivých pravidiel, ani samotnú konštrukciu derivačného stromu. Nad jednotlivými uzlami stromu vykonávame operácie a postupne vytvárame konečný automat, resp. pomocné množiny z ktorých následne skonštruujeme plnohodnotný konečný automat.

Gramatické pravidlá PCRE výrazov sú nasledujúce:

- Alternácia $V \rightarrow V \mid V$
- Konkatenácia $V \rightarrow VV$
- Kvantifikátor + $V \rightarrow V^{+}$
- Kvantifikátor * $V \rightarrow V^{*}$
- Kvantifikátor ? $V \rightarrow V?$
- Zátvorky $V \rightarrow (V)$
- Symbol $V \rightarrow s$

Pomocou týchto pravidiel generujeme derivačný strom regulárneho výrazu. Pravá časť nám vyjadruje na aký výraz sa môže rozgenerovať časť ľavá.

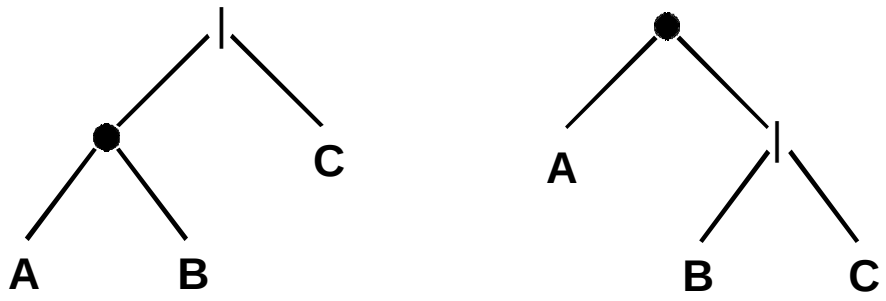
Vrchol stromu je vždy *V*, ktoré znamená výraz. *V* je neterminál. To znamená, že *V* nie je priamou súčasťou regulárneho výrazu. Terminálmi označujeme znaky, resp. operácie, ktoré sú súčasťou regulárneho výrazu, napríklad *, + alebo znak „z.“ Preto musíme aj každý jednotlivý symbol previesť na *V* pomocou pravidla $V \rightarrow s$.

Pri použití viacerých pravidiel za sebou nevieme, ktoré pravidlo pri generovaní použiť ako prvé. Na to nám slúži priorita operátorov. Znamená to, že každý operátor má danú nemennú prioritu, ktorá určuje, ktoré pravidlo máme použiť pri generovaní, ak môžeme použiť obe pravidlá.

Nech máme regulárny výraz

$$A \mid BC$$

Pomocou pravidiel môžeme vygenerovať z tohto regulárneho výrazu dva rozdielne stromy.



Z týchto jednotlivých stromov by sme skonštruovali rozdielne automaty. Na to, aby sme správne určili, ktoré pravidlo sa musí použiť, musíme poznať priority operátorov. Operátor konkatenácia má vyššiu prioritu ako alternácia. Preto ako prvý vygenerujeme podstrom z *AB*, a potom generujeme strom z *AB* a *C*. Ak majú operátory rovnakú prioritu, tak použijeme operáciu, ktorá je v regulárnom výraze najviac naľavo.

S prioritou súvisí pri konštrukcii stromu precedenčná tabuľka. Jej obsah pre regulárne výrazy si ukážeme a vysvetlíme v nasledujúcej kapitole.

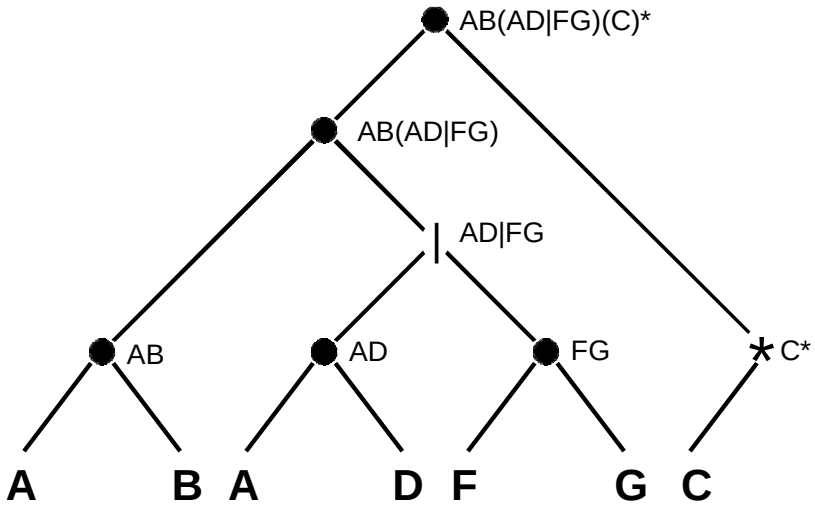
Priority jednotlivých operátor sú nasledovné:

Kvantifikátory + * ? {} > *Konkatenácia* > *Alternácia*

Pre demonštráciu viacerých pravidiel v zložitejšom regulárnom výraze si ukážeme derivačný strom pre nasledujúci výraz

$$AB(AD \mid FG)(C)*$$

Derivačný strom bude vyzerať nasledovne:

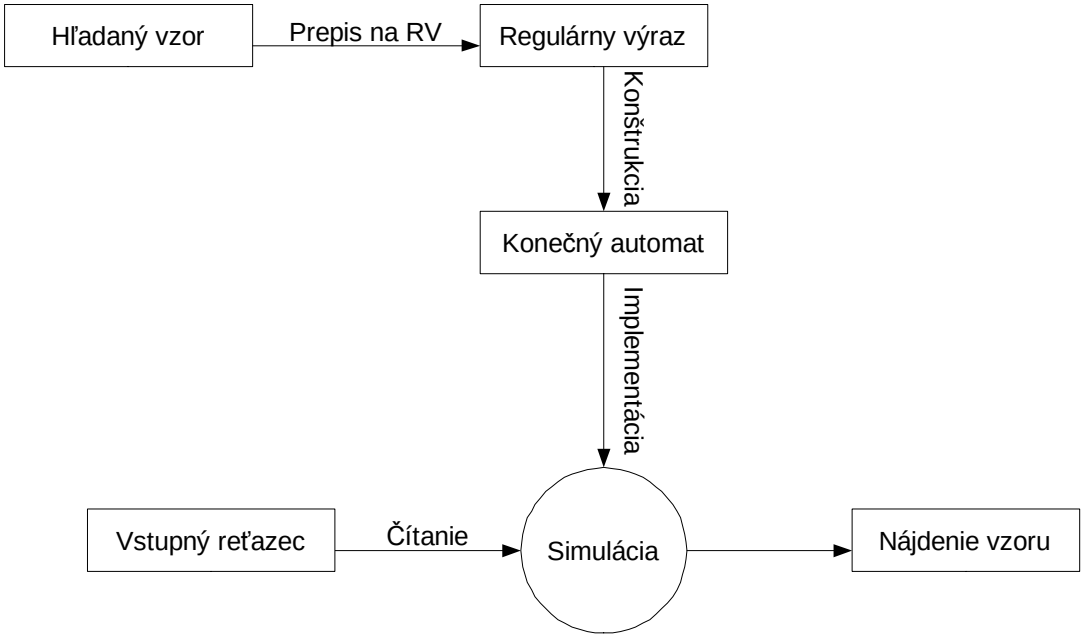


3 Nedeterministické konečné automaty

Na definíciu pravidiel pre IDS systémy používame regulárne výrazy. Najväčšie zastúpenie majú PCRE výrazy. Regulárne výrazy sa v IDS systéme interpretujú pomocou konečných automatov. Rozhodli sme sa rozobrať techniky na konštrukciu nedeterministických konečných automatov, pretože náš výskum je orientovaný na IDS systémy implementované v FPGA.

Konečný automat je graf. Jeho základom sú stavy a prechody. Pomocou konečného automatu dokážeme simulovať hľadanie vzoru v reťazci. Regulárne výrazy dokážeme previesť na konečné automaty, ktoré ich reprezentujú. Konečný automat číta vstupné symboly a pomocou prechodov, prechádza do iných stavov. Ak konečný automat prejde do stavu označeného ako koncový, tak sme našli v reťazci vzor, ktorý odpovedá regulárnemu výrazu.

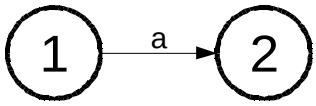
V práci sme sa zamerali na konštrukciu regulárneho výrazu a následne na zistenie informácií a konečnom automate, ktoré sa odrazia v jeho simulovaní.



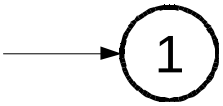
Pri grafickej reprezentácii konečného automatu zobrazujeme stav ako kruh. Stav môže byť označený unikátnym identifikátorom. V našom prípade je to číslo 1.

Prechody označujeme šípkou, ktorá vychádza zo stavu a vstupuje do stavu. Je pri nej uvedený prechodový symbol, pomocou ktorého prechádza automat medzi stavmi pri prijatí tohto symbolu

z reťazca. Prechod zapisujeme v tvare *p(s)→n*, kde *p* značí pôvodný stav, *s* je prechodový symbol a *n* je nasledujúci stav.



Začiatočný stav konečného automatu označujeme pomocou šípky ktorá doň vstupuje a nevystupuje zo žiadneho stavu. Týmto určujeme že tento stav bude prvý do ktorého vstúpime a bude spracovávať prvý symbol reťazca.



Koncové stavy zobrazujeme pomocou zvýrazneného kruhu. Ak sa pri prijímaní vstupného reťazca dostaneme do tohto stavu, znamená to, že sme v reťazci našli hľadaný vzor určený regulárnym výrazom.

Definícia nedeterministického konečného automatu je nasledovná [7]:

Konečný automat A *je pätica* $A=(Q,T,\delta,s,F)$, *kde:*

→ Q *je množina stavov*

→ T *je abeceda vstupných symbolov*

→ *R je množina pravidiel v tvare p(a)→q, kde* $p,q\in Q$; $a\in T\cup\{\varepsilon\}$

→ $s\in Q$ *je počiatočný stav*

→ $F\subseteq Q$ *je množina koncových stavov*

Každý nedeterministický konečný automat dokážeme vyjadriť pomocou týchto množín. Ak dokážeme previesť regulárny výraz na tieto množiny, tak sme schopný simulovať tento konečný automat pri prechádzaní reťazca.

Nedeterministický konečný automat, na rozdiel od deterministického, povoľuje viac ako jeden prechod s rovnakým prechodovým symbolom z jedného konkrétneho stavu. Determinizmus v názve vyjadruje, že máme presne určený maximálne jeden stav do ktorého máme s daným vstupným symbolom prejsť [5]. Pritom môže mať nedeterministický konečný automat menej stavov ako deterministický a reprezentovať ten istý regulárny výraz.

Deterministické konečné automaty je výhodnejšie použiť pri sériovom simulovaní, keď môžeme pomocou jedného symbolu prejsť iba do jedného nasledujúceho stavu. Pri simulácii nedeterministického konečného automatu by sme sa museli rekurzívne zanoriť do všetkých možných ciest vedúcich z každého nedeterministického stavu a pri neúspešnej ceste sa vrátiť naspať a prehľadávať ďalšie stavy, do ktorých sú možné prechody.

Na zostrojenie konečného automatu z regulárneho výrazu existuje niekoľko prístupov. My sa budeme zaoberať Thompsonovým a Gluškovovým konečným automatom. Každý používa iný prístup pri konštrukcii z regulárneho výrazu a taktiež má iný výstupný konečný automat.

Rozhodujúce charakteristiky regulárnych výrazov použitých v hardvérových IDS systémoch, ktoré budeme bližšie skúmať, sú nasledujúce:

- počet stavov
- počet prechodov
- výskyt ε prechodov
- rôznosť prechodov do jednotlivých stavov

Pri použití konečných automatov sú pre nás dôležité výsledné vlastnosti implementácie regulárneho výrazu v hardvéri.

Prvý dôležitý faktor je zložitosť algoritmu po implementovaní. Výhodné by bolo, ak by bol vytvorený konečný automat bez epsilon prechodov. Každý konečný automat dokážeme previesť na automat bez epsilon prechodov, tento prevod nás však stojí úsilie vynaložené navyše.

Druhým faktorom je veľkosť vytvoreného kódu, ktorý bude simulovať činnosť konečného automatu. V tomto prípade preferujeme automat s ohľadom na čo najmenší počet stavov a prechodov.

Tretím faktorom je zložitosť implementácie v hardvéri. Zjednodušenie architektúry implementácie nám prispeje k zmenšeniu plochy použitej na čipe.

3.1 Thompsonov konečný automat

Thompsonov konečný automat je priamou reprezentáciou derivačného stromu regulárneho výrazu. Pre zjednodušenie konštrukcie využíva epsilon prechody.

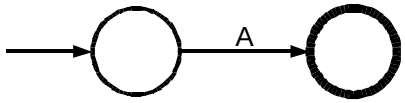
Pre každý podstrom derivačného stromu vytvárame samostatný konečný automat. Pri konštrukcii spájame jednotlivé podstromy od listov až po najvyšší uzol. Každý obsahuje dva pomocné stavy, do ktorých, resp. z ktorých smerujú epsilon prechody.

Prvým pomocným stavom je počiatočný stav, z ktorého sa pomocou epsilon prechodov môžeme dostať do všetkých následných stavov. Druhým stavom je koncový stav podstromu, do ktorého sa môžeme dostať epsilon prechodmi zo všetkých koncových stavov podstromu.

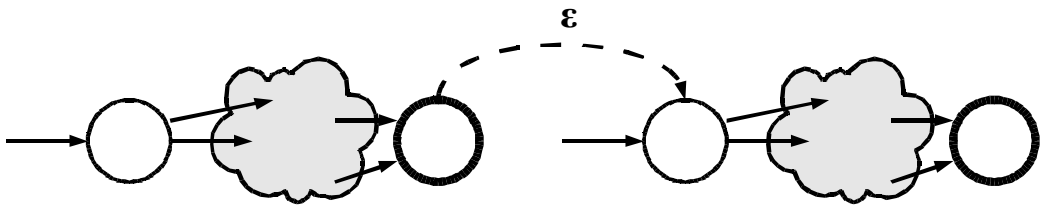
Pomocou týchto dvoch stavov môžeme relatívne jednoducho aplikovať rôzne operácie na jednotlivé podstromy. Každý podstrom obsahuje vždy iba jeden začiatočný a jeden koncový stav.

Pravidlá pre konštrukciu konečného automatu z regulárneho výrazu sú nasledovné:

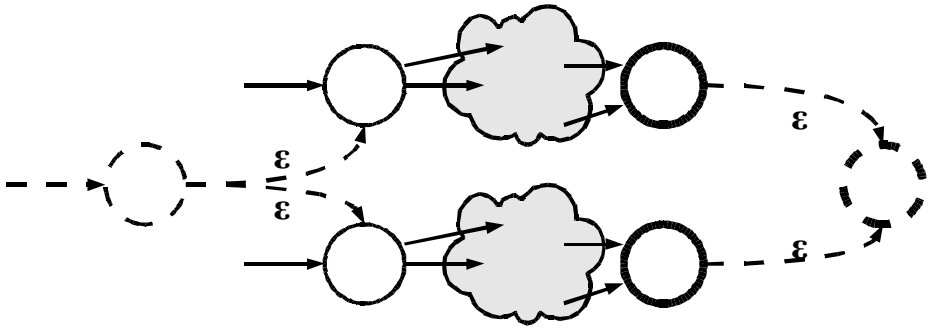
Pre jednoduchý znak *A* vytvoríme tri stavy a dva prechody. Začiatočný a koncový stav, do ktorých smerujú epsilon prechody a stav a prechod, ktorý bude vyjadrovať prijatie znaku *A*.



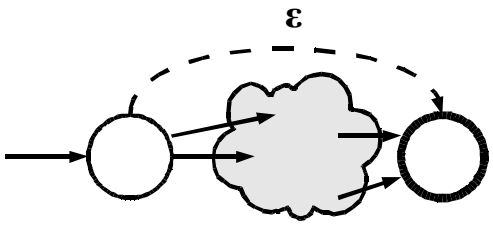
Pri konkatenácii dvoch výrazov vytvoríme epsilon prechod z koncového stavu konečného automatu do začiatočného stavu koncového automatu.



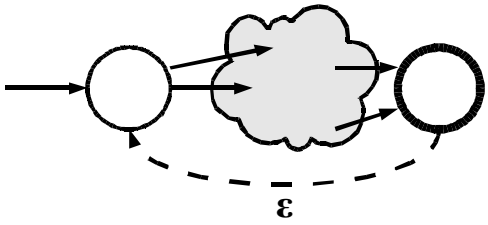
Pre alternáciu vytvoríme začiatočný a koncový stav. Vytvoríme epsilon prechody zo začiatočného do začiatočných stavov oboch výrazov. Následne vytvoríme taktiež epsilon prechody z koncových stavov jednotlivých výrazov do nového koncového stavu.



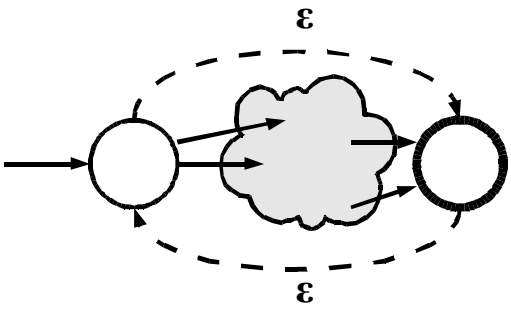
Kvantifikátor ? vyjadríme pomocou jedného epsilon prechodu. Tento nám značí, že regulárny výraz môžeme, ale nemusíme prijať. Možnosť prijatia jedenkrát nám spĺňa konečný automat samotný. Vytvoríme preto epsilon prechod zo začiatočného stavu do koncového, ktorý nám vyjadrí neprijatie vnútorného konečného automatu.



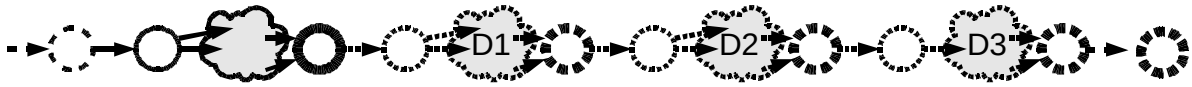
Kvantifikátor + taktiež vyjadríme jedným epsilon prechodom. Použitie tohto kvantifikátoru znamená, že regulárny výraz, na ktorý je použitý, musíme prijať minimálne jedenkrát. Možnosť prijatia jedenkrát nám spĺňa konečný automat samotný. Vytvoríme epsilon prechod z koncového stavu do začiatočného stavu, ktorý nám po prijatí regulárneho výrazu dovolí použiť tento výraz znovu.



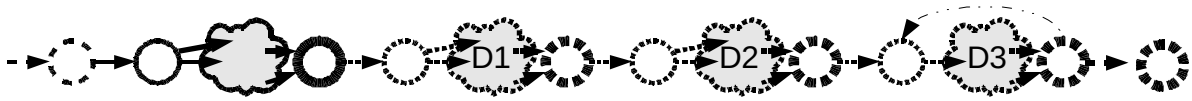
Kvantifikátor * vyjadríme dvomi epsilon prechodmi. Vyjadruje, že regulárny výraz, na ktorý je použitý, sa môže opakovať ľubovoľne krát. Tento kvantifikátor v sebe kombinuje kvantifikátory + a *. Práve preto využijeme oba spôsoby konštrukcie týchto kvantifikátorov. Vytvoríme epsilon prechod z koncového stavu do začiatočného a taktiež epsilon prechod zo začiatočného do koncového.



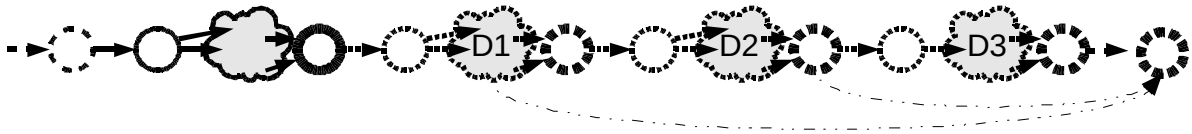
Kvantifikátor {m} značí, že regulárny výraz, na ktorý sa viaže musíme prijať nie iba raz, ale *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou. Vytvoríme epsilon prechod z koncového stavu pôvodného podstromu do začiatočného stavu prvého zduplikovaného konečného automatu, z nového prvého stavu stromu do začiatočného stavu pôvodného podstromu a z koncového stavu posledného zduplikovaného konečného automatu do posledného stavu stromu.



Kvantifikátor {m,} značí, že regulárny výraz, na ktorý sa viaže musíme prijať minimálne *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou. Vytvoríme epsilon prechod z koncového stavu pôvodného podstromu do začiatočného stavu prvého zduplikovaného konečného automatu, z nového prvého stavu stromu do začiatočného stavu pôvodného podstromu a z koncového stavu posledného zduplikovaného konečného automatu do posledného stavu stromu. Nakoniec vytvoríme epsilon prechod na poslednom zduplikovanom automate, z koncového stavu do je počiatočného stavu.



Kvantifikátor {m,n} značí, že regulárny výraz, na ktorý sa viaže musíme prijať nie iba raz, ale minimálne *m-krát* a maximálne *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou. Vytvoríme epsilon prechod z koncového stavu pôvodného podstromu do začiatočného stavu prvého zduplikovaného konečného automatu, z nového prvého stavu stromu do začiatočného stavu pôvodného podstromu a z koncového stavu posledného zduplikovaného konečného automatu do posledného stavu stromu. Z koncových stavov duplikovaných automatov z rozsahu $\langle m-1, n-1 \rangle$ vytvoríme epsilon prechody do koncového stavu stromu. Rozsah je posunutý, pretože prvý stav je neduplikovaný a má poradové číslo 0.



Pretože Thompsonov konečný automat po konštrukcii z regulárneho výrazu obsahuje epsilon prechody, musíme použiť techniku na ich odstránenie. Po tejto transformácii získame konečný automat, ktorý môžeme použiť na simuláciu hľadania vzoru v hardvéri.

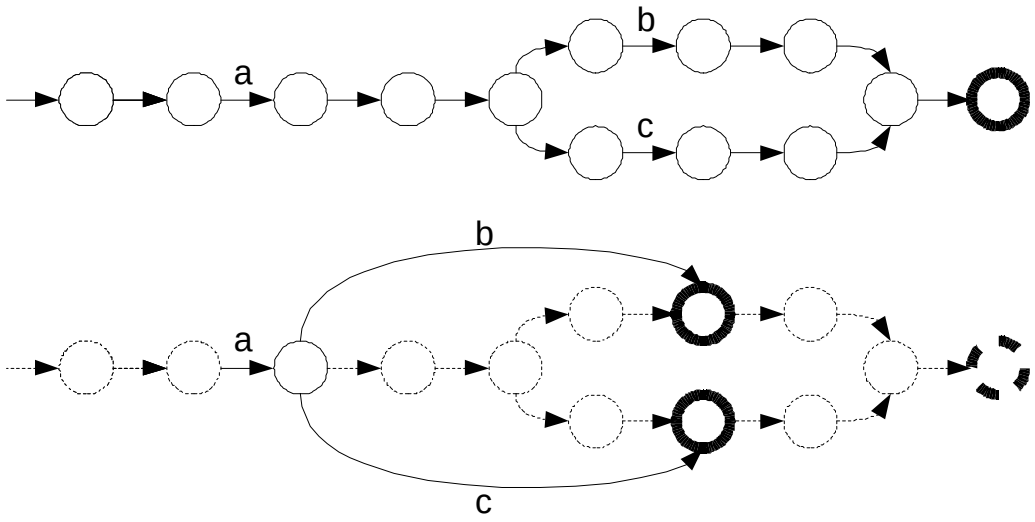
Epsilon prechod nám značí, že konečný automat môže prejsť pomocou tohto prechodu medzi stavmi bez prečítania vstupného symbolu. Implementovať epsilon prechody v simulácii by bolo zložité a pomalé.

Proces odstránenie epsilon prechodov je nasledovný:

Zo všetkých stavov, do ktorých vstupujú aj iné ako epsilon prechody, tzn. do stavov, do ktorých sa môžeme dostať aj pomocou prijímaných znakov, vytvoríme nové prechody. Nové prechody získame pomocou rekurzívneho prechádzania všetkých možných ciest cez epsilon prechody, ak narazíme na ne-epsilon prechod, tak vytvoríme nový prechod. Stavy, do ktorých sa

môžeme pomocou epsilon prechodov dostať zo začiatočných stavov sa taktiež stanú začiatočnými stavmi. Stavy z ktorých sa môžeme pomocou epsilon prechodov dostať do koncových stavov sa taktiež stanú koncovými stavmi. Tento proces by sme mohli popísať tiež pomocou epsilon-uzáveru, ale použijeme ho pri porovnaní Gluškovovho a Thompsonovho automatu v kapitole 5.

Následne odstránime všetky epsilon prechody. Potom odstránime stavy, do ktorých, resp. z ktorých nevedú žiadne prechody.



3.2 Gluškovov konečný automat

Gluškovov konečný automat je produkt súboru pravidiel, ktorými určujeme ako konečný automat zostrojiť. Tieto pravidlá pomenujme Gluškovov algoritmus. Je charakteristický tým, že obsahuje maximálne *n+1* stavov, kde *n* je počet znakov v regulárnom výraze. Ďalšou zaujímavou vlastnosťou je fakt, že do každého stavu môžu smerovať iba prechody s jedným rovnakým vstupným symbolom. Gluškovov konečný automat neobsahuje epsilon prechody, takže sa nemusíme starať o prevod na konečný automat bez epsilon prechodov.

Prvým krokom pre zostrojenie konečného automatu z regulárneho výrazu pomocou Gluškovovho algoritmu je označenie jednotlivých znakov v regulárnom výraze.

Nech máme regulárny výraz

$$AB(AD \mid FG)(C)*$$

Každému znaku priradíme jedinečné číslo počínajúc číslom 1 a končiac n (celkový počet znakov). Označený regulárny výraz bude vyzerať nasledovne

$$A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*$$

Každý označený znak nám reprezentuje konkrétny stav konečného automatu. Vstupným znakom pre každý stav je znak, ktorý ho reprezentuje. Vďaka tomu môžu do jedného konkrétneho stavu viesť prechody iba s jedným vstupným znakom.

Pri konštruovaní konečného automatu si vytvoríme tri pomocné množiny: *First(), Last()* a *Follow()*.

Množina *First(RV)* obsahuje stavy regulárneho výrazu *RV* do ktorých sa môžeme dostať po prijatí prvého znaku vstupného reťazca. V našom príklade:

$$First(A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*) = \{A_1\}.$$

Množina *Follow(RV, x)* obsahuje možné následné stavy regulárneho výrazu *RV* zo stavu *x*. V našom príklade:

$$Follow((A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*), B_2) = \{A_3, F_5\}$$

Množina *Last(RV)* obsahuje koncové stavy konečného automatu vytvoreného z regulárneho výrazu *RV*. V našom príklade:

$$Last(A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*) = \{D_4, G_6, C_7\}$$

Množiny *First(), Last()* a *Follow()* musíme previesť na konečný automat v tvare definovanom na začiatku kapitoly.

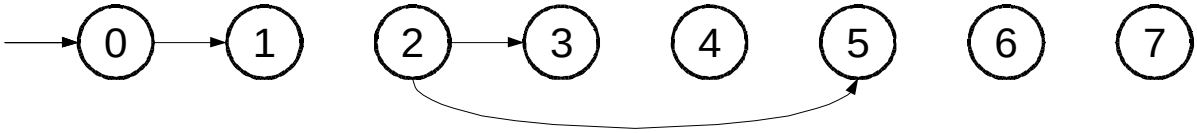
K existujúcim stavom si vytvoríme špeciálny stav 0. Tento bude reprezentovať prvý vstupný stav konečného automatu.

Z tohto stavu si vytvoríme prechody do všetkých stavov z množiny *First()*. V našom prípade je to iba jeden stav.

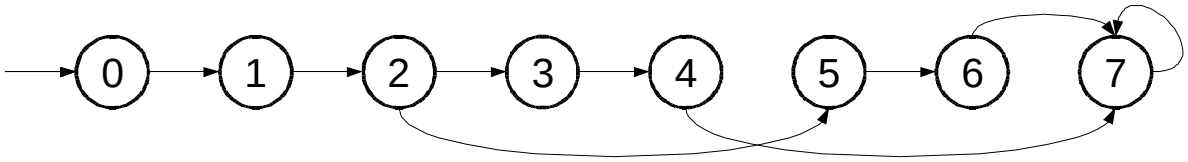
$$First(A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*) = \{A_1\}$$

Ďalej z jednotlivých stavov, pomocou množiny *Follow()* vytvoríme prechody do nasledujúcich stavov. Názorne si to ukážeme na stave 2.

$$Follow((A_1B_2(A_3D_4 \mid F_5G_6)(C_7)*), B_2) = \{A_3, F_5\}$$

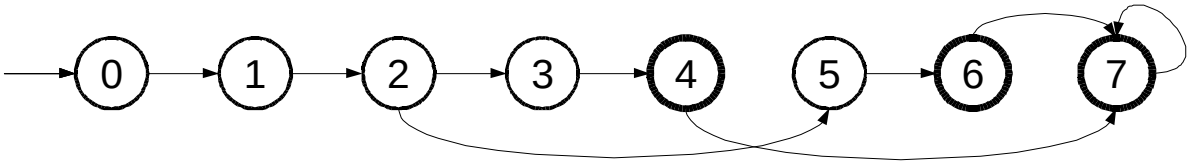


Toto urobíme pre všetky stavy. Nakoniec nám vznikne takýto konečný automat:

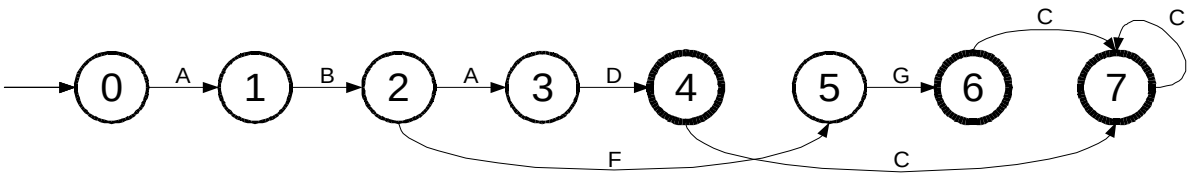


Podľa obsahu množiny *Last()* si vyznačíme koncové stavy.

$$Last(A_1 B_2 (A_3 D_4 \mid F_5 G_6)(C_7)^*) = \{D_4, G_6, C_7\}$$



Aby bol konečný automat kompletný, musíme si pomenovať jednotlivé prechody. Pre každý stav označíme všetky prechody doňho vstupujúce znakom, ktorý tento stav vyjadruje v regulárnom výraze.



Po predchádzajúcich transformáciách je tento konečný automat plnohodnotným automatom pre použitie v IDS systémoch.

Výsledné množiny konečného automatu pre tento regulárny výraz sú nasledujúce:

$$Q = \{1,2,3,4,5,6,7\} \qquad s = \{0\} \qquad F = \{4,6,7\}$$

$$T = \{0(A) \to 1,1(B) \to 2,2(A) \to 3,2(F) \to 5,3(D) \to 4,4(C) \to 7,5(G) \to 6,6(C) \to 7,7(C) \to 7\}$$

4 Implementácia

V tejto kapitole si detailnejšie rozoberieme konkrétnu implementáciu oboch typov konečných automatov. Bližšie si rozoberieme prevod regulárneho výrazu na derivačný strom. Ukážeme si regulárne výrazy, ktoré sme implementovali a výrazy, ktoré sme sa rozhodli z rôznych dôvodov neimplementovať.

Program sme sa rozhodli urobiť objektovo orientovaný. Preto sme použili jazyk C++, ktorý nám dovolil objektovú implementáciu a použitie rozličných dátových typov.

Proces prevodu regulárneho výrazu na konečný automat má pri implementácii tri fázy. Je to regulárny výraz, ktorý následne prevedieme na reprezentáciu konečným derivačným stromom. Z derivačného stromu nakoniec vytvoríme konečný automat.

4.1 Podporovaná gramatika

Dokázali sme implementovať nasledujúce gramatické pravidlá: konkatenácia, alternácia, kvantifikátory +,*,?, {m}, {m,}, {m,n} a triedy znakov.

Nepodporujeme spätné referencie, z dôvodov ich náročnej implementácie. Previesť spätné referencie na konečný automat je veľmi zložitá úloha, pretože sa v nich odkazujeme na reťazce prijaté v predchádzajúcej časti výrazu. Taktiež nepodporujeme konštrukcie typu (?) z dôvodov ich veľkej rozmanitosti a počtu operácií, ktoré môžeme pomocou nich vyjadriť.

4.2 Lexikálna analýza

Lexikálny analyzátor je deterministický konečný automat, ktorý rozpoznáva jednotlivé lexémy regulárneho výrazu. Konečný automat číta vstupný reťazec a pri zhode vygeneruje dátovú štruktúru, ktorá reprezentuje daný lexém v podobe zrozumiteľnej pre syntaktický analyzátor.

Pre uľahčenie práce syntaktického analyzátoru preberá niektoré jeho jednoduché funkcie. Keďže operácia konkatenácie nie je vyjadrená žiadnym metaznakom, tak lexikálny analyzátor tento metaznak virtuálne vytvorí na miestach, kde by sa mal nachádzať a predá ho syntaktickému analyzátoru v podobe lexému.

Ak sa v regulárnom výraze nachádza lexikálna chyba, analyzátor je schopný ju odhaliť a patrične zareagovať.

4.3 Syntaktická analýza

Vstupom syntaktickej analýzy sú lexémy z lexikálneho analyzátoru a precedenčná tabuľka pre danú gramatiku. Výstupom je derivačný strom z regulárneho výrazu.

Pri vyhodnocovaní výrazov sme zvolili techniku „zdola – nahor“ s využitím zásobníka. Pri práci so zásobníkom sme použili precedenčnú tabuľku. Táto tabuľka nám pomáha pri syntaktickej analýze v rozhodovaní ako narábať s vstupnými symbolmi a ukladaním ich na zásobník, resp. s jeho redukovaním.

Algoritmus analýzy „zdola - nahor“ je nasledovný [7]:

- Vlož na zásobník znak $ (značí začiatok, resp. koniec regulárneho výrazu – ak lexikálny analyzátor prečíta celý regulárny výraz, tak vráti $, tento znak nemá súvis s PCRE metaznakom $)
- Nech je *t* = najbližšia operácia, zátvorka alebo $ najbližší k vrcholu zásobníku
  Nech je *a* = aktuálny načítaný lexém
- Ak (*t* == *a*) tak je derivačný strom úspešne vytvorený, koniec
- Case Tabuľka[*a*, *t*]:
  - = : redukuj na zásobníku zátvorky (pravidlo „*V→(V)*“)
  - < : vlož *a* na vrchol zásobníku a načítaj nové *a*
  - > : zredukuj pravidlo na zásobníku a vykonaj danú operáciu, zároveň spracujeme uzol a vytvoríme nové stavy a prechody
  - X: syntaktická chyba, koniec
- Vráť sa na bod 2

Precedenčná tabuľka pre PCRE gramatiku:

	t	**|**	**.**	**+**	*****	**?**	**{}**	**(**	**)**	**i**	**$**
a											
|		**>**	**<**	**<**	**<**	**<**	**<**	**<**	**>**	**<**	**>**
.		**>**	**>**	**<**	**<**	**<**	**<**	**<**	**>**	**<**	**>**
+		**>**	**>**	**>**	**>**	**>**	**>**	**X**	**>**	**X**	**>**
*****		**>**	**>**	**>**	**>**	**>**	**>**	**X**	**>**	**X**	**>**
?		**>**	**>**	**>**	**>**	**>**	**>**	**X**	**>**	**X**	**>**
(		**<**	**<**	**<**	**<**	**>**	**>**	**<**	**=**	**X**	**<**
)		**>**	**>**	**>**	**>**	**>**	**>**	**X**	**<**	**X**	**>**
i		**>**	**>**	**>**	**>**	**<**	**<**	**X**	**>**	**X**	**>**
$		**<**	**<**	**<**	**<**	**>**	**>**	**X**	**<**	**X**	**<**

4.4 Konštrukcia konečného automatu z derivačného stromu

Po vytvorení derivačného stromu nám už iba zostáva previesť ho na konečný automat. Tento prevod je rozdielny pre naše prístupy, ktoré sme si zvolili. Vo všeobecnosti majú oba prístupy rovnaké črty v tom, že prechody vznikajú pri spracovávaní uzlov, ktoré reprezentujú základné operácie. Taktiež začiatočné a koncové stavy nám vznikajú až na vrchole derivačného stromu. Rozdielne črty sa objavujú pri vytváraní jednotlivých stavov konečného automatu. Pri Gluškovovom algoritme ich vytvárame už pri lexikálnej analýze (neplatí pre kvantifikátory {m,n}, {m,} a {m}). Pri Thompsonovom prístupe tvoríme stavy až pri konštrukcii z derivačného stromu, pretože nevieme dopredu určiť, koľko stavov bude potrebných.

Konštrukcia pri oboch prístupoch je založená na skutočnosti, že pre každý podstrom derivačného stromu, môžeme vytvoriť konečný automat, ktorý ho bude reprezentovať. Každý podstrom má teda vlastné množiny počiatočných a koncových stavov. Pomocou týchto množín môžeme jednotlivé podstromy spájať aplikovaním operácií.

4.4.1 Gluškovov nedeterministický konečný automat

Pri Gluškovovom algoritme, si najprv vytvoríme Množiny *First(), Last()* a *Follow(),* ktoré následne prevedieme na plnohodnotný konečný automat. Tento prevod a význam množín sme si ukázali v kapitole 3.2. Teraz si ukážeme ako vytvoríme aj jednotlivé prechody, resp. prvky množiny *Follow().*

Pre každý samostatný podstrom derivačného stromu existujú množiny *First(), Last()* a *Follow().*

Pre všetky podstromy platí

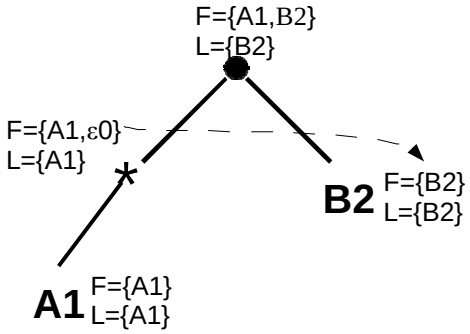
$\mathrm{Follow}(strom) = \{Follow(podstrom1) \cup Follow(podstrom2) \cup ... \cup Follow(podstromN)\}$, *kde N je počet podstromov stromu.*

Preto nemusíme pre každý podstrom vytvárať samostatnú množinu *Follow(),* ale môžeme použiť spoločnú množinu pre celý regulárny výraz. To znamená, že môžeme napĺňať túto množinu počas celej konštrukcie z derivačného stromu, bez ohľadu na to, ktorý podstrom spracovávame.

Lexikálny analyzátor nám pri čítaní regulárneho výrazu uloží do tabuľky informácie o načítaných vstupných symboloch a priradí im jednoznačný identifikátor, v našom prípade je to číslovanie od 1. Načítané symboly sú vždy uzlami stromu.

Pri konštrukcii používame v množinách *First()* a *Last()* špeciálny virtuálny stav ε_0. Je to stav, pomocou ktorého sa môžeme dostať virtuálnym epsilon prechodom do nasledujúceho stavu. Ukážeme si jeho funkciu na nasledujúcom príklade:

Nech máme regulárny výraz *A*B*. Derivačný strom vyzerá nasledovne:

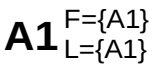


Použitie operácie * na znak A znamená, že znak sa môže opakovať ľubovoľne krát. Teda znak A sa nemusí opakovať vôbec, tzn. že v množine *First()* sa nám nemusí vyskytovať iba stav A1, ale môže tam byť stav do ktorého vstupujeme z množiny *Last()*. Pri použití operácie * ešte nepoznáme stav, do ktorého sa môžeme dostať, preto použijeme stav ε_0. Pri konkatenácii teda nahradíme stav ε_0 z ľavého podstromu za všetky stavy z množiny *First()* z pravého podstromu.

Vďaka stavu ε_0 nemusíme používať epsilon prechody, ktoré nám nedovoľuje Gluškovov prístup vytvárať. Použitie stavu ε_0 v množine *First()* znamená, že tento podstrom nemusí prijať žiadny symbol. ε_0 nahradíme za stavy, do ktorých nám vzniknú prechody z množiny *Last()*. Pri použití konkatenácie môžeme vytvoriť prechod zo stavu *A1* do stavu *B2*, takže ε_0 nahradíme za *B2*.

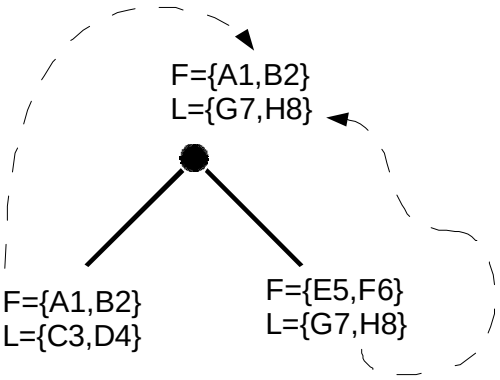
Algoritmy spracovania jednotlivých operácií a vytvorenia množín vyzerajú nasledovne:

Pri liste stromu, teda znaku z reťazca vyvárame množiny podľa tohto znaku. Vo *First()* aj v *Last()* bude znak samotný. Nevznikajú nám žiadne prvky množiny *Follow()*.



Pre zjednodušenie vysvetľovania nasledujúcich pravidiel si množiny *First()* a *Last()* jednotlivých podstromov označme ako *leftF*, *leftL*, *leftF* a *rightL*, kde *left* a *right* určujú množiny ľavého, resp. pravého podstromu.

Operácia konkatenácie nám vytvorí prechody zo všetkých prvkov množiny *leftL* do všetkých prvkov množiny *rightF*. Množiny nového stromu budú *First()={leftF}, Last()={rightL}*.

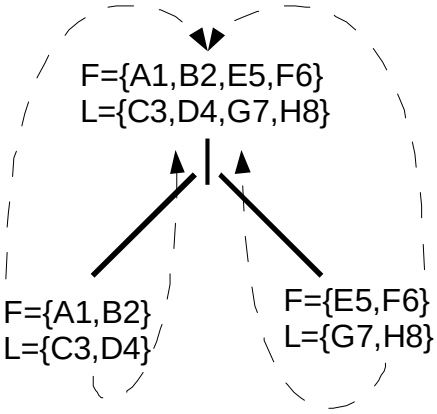


Vzniknú nám tieto prechody:

C3→E5 *C3→F6*

D4→E5 *D4→F6*

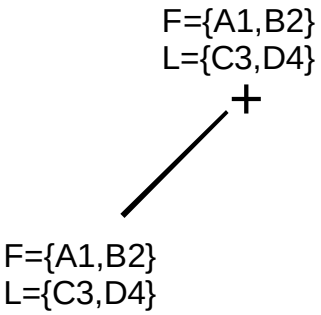
Pri aplikovaní alternácie nám nevznikajú žiadne prechody. Množiny First() a Last() sú zjednotením týchto množín z oboch podstromov.



$$First(\ ) = \{leftFirst \cup rightFirst\}$$
$$Last(\ ) = \{leftLast \cup rightLast\}$$

Nasledujúce operácie obsahujú vždy len jeden podstrom. Použijeme preň označenie *left*.

Kvantifikátor + nám vytvorí prechody zo všetkých prvkov množiny *leftL* do všetkých prvkov množiny *leftF*. Po aplikovaní tejto operácie zostávajú množiny *First()* a *Last()* nezmenené.



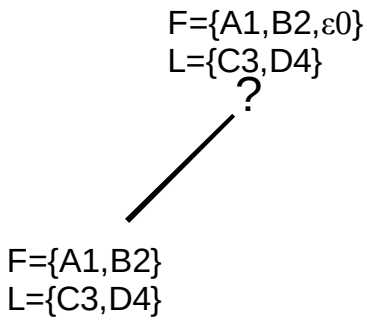
Vzniknú nám nasledovné prechody

C3→A1 *D4→A1*

C3→B2 *D4→B2*

$$First(\ ) = \{leftFirst\}$$
$$Last(\ ) = \{leftLast\}$$

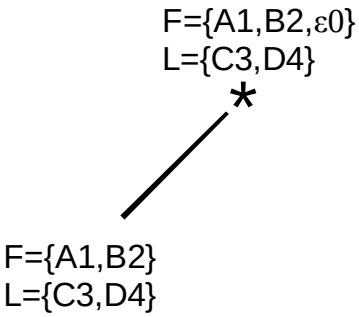
Kvantifikátor ? nevytvára žiadne prechody. Po aplikovaní tejto operácie zostáva množina *Last()* nezmenená. Do množiny *First()* pridáme epsilon stav ε_0.



Pri tejto operácii nám nevznikajú prechody.

$$First(\ ) = \{leftFirst \cup \varepsilon_0\}$$
$$Last(\ ) = \{leftLast\}$$

Kvantifikátor * nám vytvorí prechody zo všetkých prvkov množiny *leftL* do všetkých prvkov množiny *leftF*. Po aplikovaní tejto operácie zostáva množina *Last()* nezmenená. Do množiny *First()* pridáme epsilon stav ε_0.



Vzniknú nám nasledovné prechody

C3→A1 *D4→A1*

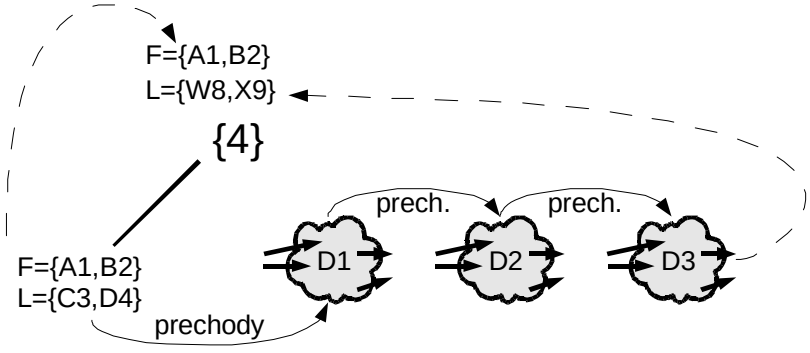
C3→B2 *D4→B2*

$$First(\ ) = \{leftFirst \cup \varepsilon_0\}$$

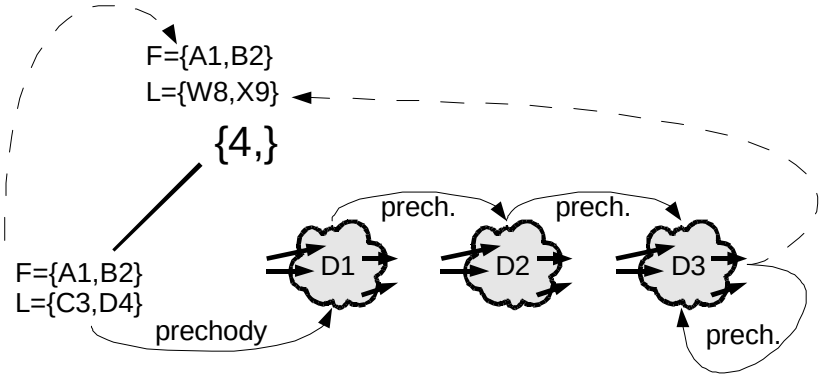
$$Last(\ ) = \{leftLast\}$$

Kvantifikátory typu {m,n}, {m,} a {m} sú operácie, ktoré nemôžeme aplikovať bez porušenia pravidiel konštrukcie Gluškovovho algoritmom. Pri týchto operáciách musíme vytvoriť stavy na viac, čím porušíme pravidlo, že Gluškovov konečný automat má *n+1* stavov, kde *n* je počet symbolov v regulárnom výraze. Pri všetkých kvantifikátoroch tohto typu niekoľkokrát zduplikujeme konečný automat podstromu a potom do počiatočných, respektíve z koncových stavov vytvárame nové prechody.

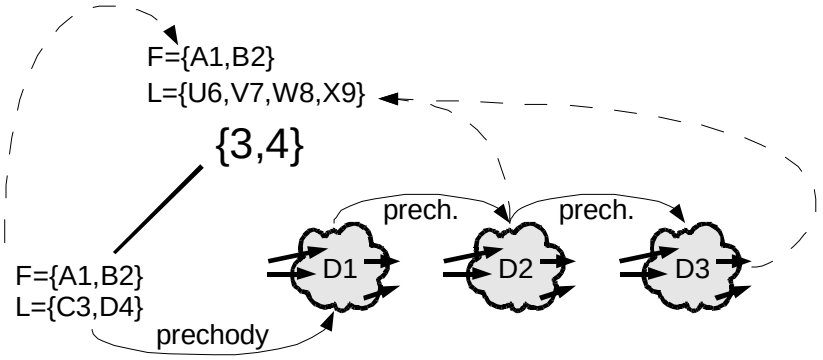
Kvantifikátor {m} značí, že regulárny výraz, na ktorý sa viaže musíme prijať nie iba raz, ale *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou a vytvoríme medzi nimi prechody. Nakoniec vytvoríme prechody z leftL do prvého zduplikovaného konečného automatu. Koncovými stavmi sa stanú koncové stavy posledného duplikovaného konečného automatu.



Kvantifikátor {m,} značí, že regulárny výraz, na ktorý sa viaže musíme prijať minimálne *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou a vytvoríme medzi nimi prechody. Početnosť „*minimálne n-krát*“ vytvoríme prechodmi z množiny *Last()* posledného duplikovaného konečného automatu do množiny *First()* toho istého automatu. Nakoniec vytvoríme prechody z leftL do prvého zduplikovaného konečného automatu. Koncovými stavmi sa stanú koncové stavy posledného duplikovaného konečného automatu.



Kvantifikátor {m,n} značí, že regulárny výraz, na ktorý sa viaže musíme prijať nie iba raz, ale minimálne *m-krát* a maximálne *n-krát*. Vytvoríme si *n-1* duplicitných konečných automatov a postupne ich pospájame konkatenáciou a vytvoríme medzi nimi prechody. Nakoniec vytvoríme prechody z leftL do prvého zduplikovaného konečného automatu. Množina *Last()* bude pozostávať z koncových stavov zduplikovaných konečných automatov s poradovým číslom v rozsahu $\langle m-1, n-1 \rangle$. Rozsah je posunutý, pretože prvý stav je neduplikovaný a má poradové číslo 0.



Pomocou tohto algoritmu získame v najvyššom vrchole stromu množiny *First()*, *Last()*. Zároveň sa nám počas celej konštrukcie napĺňa množina *Follow()* do ktorej ukladáme jednotlivé prechody. Po získaní všetkých množín ich môžeme previesť na plnohodnotný konečný automat.

4.4.2 Thompsonov nedeterministický konečný automat

Pri Thompsonovom prístupe zostrojovania konečného automatu generujeme priamo konečný automat, takže si nemusíme pomáhať pomocnými množinami ako pri Gluškovovom algoritme. Jednotlivé stavy vznikajú až pri konštrukcii. Vďaka použitiu epsilon prechodov je zložitosť algoritmu konštrukcie konečného automatu z derivačného stromu jednoduchšia ako pri Gluškovovom prístupe.

Platí, že pre každý podstrom môžeme vytvoriť konečný automat, ktorý na rozdiel od Gluškovovho automatu, má iba jeden začiatočný a jeden koncový stav. Do týchto stavov vstupujú epsilon prechody a slúžia na zjednodušenie práce s pravidlami.

Algoritmus implementácie je takmer zhodný s postupom uvedeným v predchádzajúcej kapitole, preto sme sa rozhodli nepopisovať ho druhý krát.

4.5 Práca s programom

Program pozostáva z troch častí určených pre užívateľa. Každá časť dokáže spracovať súbor pravidiel z konkrétnej aplikácie, L7-Filter, Bro IDS a Snort. Následne sú vybrané regulárne výrazy predané modulu pre spracovanie regulárneho výrazu a ten zapíše výsledky do súboru. Podrobnejšie informácie o použití sú dostupné v readme súbore v prílohe na DVD spolu so zdrojovými súbormi.

4.6 Kontrola správnosti

Program na vstupe prečíta regulárny výraz a vytvorí dva konečné automaty. Jeden pomocou Gluškovovho algoritmu a druhý pomocou Thompsonovho prístupu. Každý automat vyjadruje spoločný regulárny výraz, ale ich množiny nie sú rovnaké.

Na kontrolu sme použili sadu programov FSM library vo verzii 3.7 [9], ktorá dokáže pracovať s konečnými automatmi a dokáže na ne aplikovať rôzne operácie. Pomocou FSM library sme oba konečné automaty zdeterminizovali, zminimalizovali a odstránili epsilon prechody. Následne sme mohli tieto konečné automaty porovnať. Ak by tieto konečné automaty neboli rovnocenné, tak je chyba v našom programe pri konštrukcii jedného z automatov.

5 Testovanie a výsledky testov

V tejto kapitole si ukážeme akým spôsobom sme testovali vytvorené konečné automaty. Spomenieme si programy, ktoré nám pomohli získať informácie o konečných automatoch. Pozrieme sa na množinu PCRE výrazov, ktorú sme použili ako referenčnú pri testovaní.

Pozrieme sa na výsledky štatistík z pravidiel jednotlivých IDS systémov a porovnáme ich navzájom. Z výsledkov vyvodíme závery, ktorý postup pri konštruovaní je najvhodnejší.

5.1 Vstupné výrazy

Celkovo sme skúmali regulárne výrazy z troch aplikácií určených pre detekciu vzorov v sieťovej komunikácii. Ako referenčné testovacie výrazy sme použili voľne dostupné pravidlá L7-filter pre aplikáciu Netfilter, určený pre Linux, ktoré dokáže používať aj viacero iných aplikácií [8]. Ďalším testovaným systémom bol Bro IDS [10], ktorý dokáže využívať spracovať aj pravidlá z programu Snort. Posledné testované výrazy boli voľne dostupné výrazy z aplikácie Snort [11].

Každá aplikácia používa odlišnú sadu PCRE výrazov. To znamená, že výsledné štatistiky budú rozdielne a taktiež budú mať iné parametre. Preto sme sa rozhodli urobiť štatistiky pre každú aplikáciu zvlášť, a následne ich porovnať navzájom.

Vo všetkých sadách výrazov sme našli duplicitné pravidlá. Ak sme narazili na výraz, ktorý sme už v minulosti spracovávali, tak sme ho do štatistík nezapočítali, teda sme spracovávali iba jedinečné výrazy. Počet duplicitných pravidiel bol vcelku výrazný. Napríklad pri výrazoch z aplikácie Snort sme takto zredukovali asi 16340 výrazov na približne 1940.

5.2 Testované parametre

Na konečných automatoch nás zaujímali tri základné parametre (počet začiatočných stavov sa nám bohužiaľ pomocou FSM library nepodarilo zistiť):

- počet prechodov
- počet stavov
- počet koncových stavov

Pomocou knižnice FSM library (verzia 3.7) sme boli schopní zistiť tieto informácie o konečných automatoch a následne ich porovnať. Rozhodli sme sa porovnávať vlastnosti nasledovných konečných automatov (uvádzame legendu skratiek, ktoré sme použili v grafoch a tabuľkách):

- **GNKA** Gluškovov nedeterministický konečný automat
- **TNKA** Thompsonov nedeterministický konečný automat

- **eTNKA** Thompsonov nedeterministický konečný automat bez epsilon prechodov
- **MDKA** Minimalizovaný deterministický konečný automat

Zaujímavé výsledky by mohla priniesť minimalizácia niektorého s nedeterministických konečných automatov, ale knižnica FSM library dokáže minimalizovať iba deterministické konečné automaty bez epsilon prechodov, čo sú vlastnosti, ktoré nespĺňa ani Thompsonov a ani Gluškovov prístup.

Tieto parametre by nám mali pomôcť pri výbere algoritmu na konštrukciu konečných automatov z regulárnych výrazov. Vďaka nim môžeme porovnať jednotlivé prístupy a rozhodnúť sa, ktorý je pre nás najvýhodnejší.

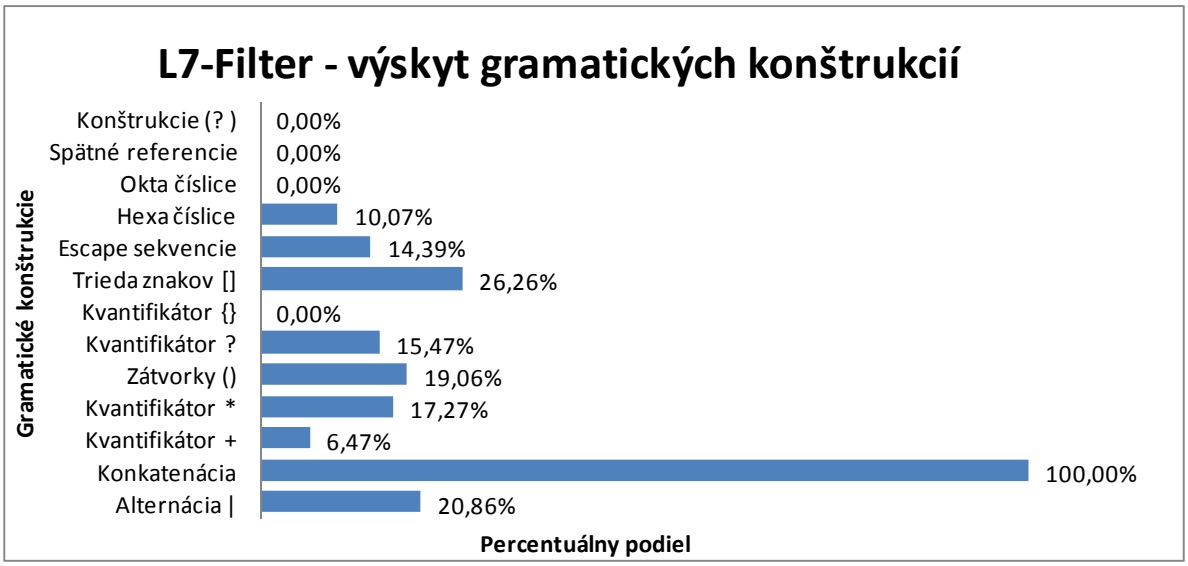
Taktiež sme sa rozhodli zmapovať výskyt jednotlivých gramatických konštrukcií. Tieto štatistiky by mali pomôcť definovaní rozsahu implementácie lexikálneho i syntaktického analyzátoru. Sledovali sme výskyt týchto gramatických konštrukcií:

- Spätné referencie
- Oktalové číslice
- Hexadecimálne číslice
- Escape sekvencie
- Triedy znakov
- Kvantifikátory {m,n}, {m,}, {m}, ?, * a +
- Zátvorky
- Konkatenácia
- Alternácia
- Konštrukcie typu (?)

5.3 Jednotlivé výsledky

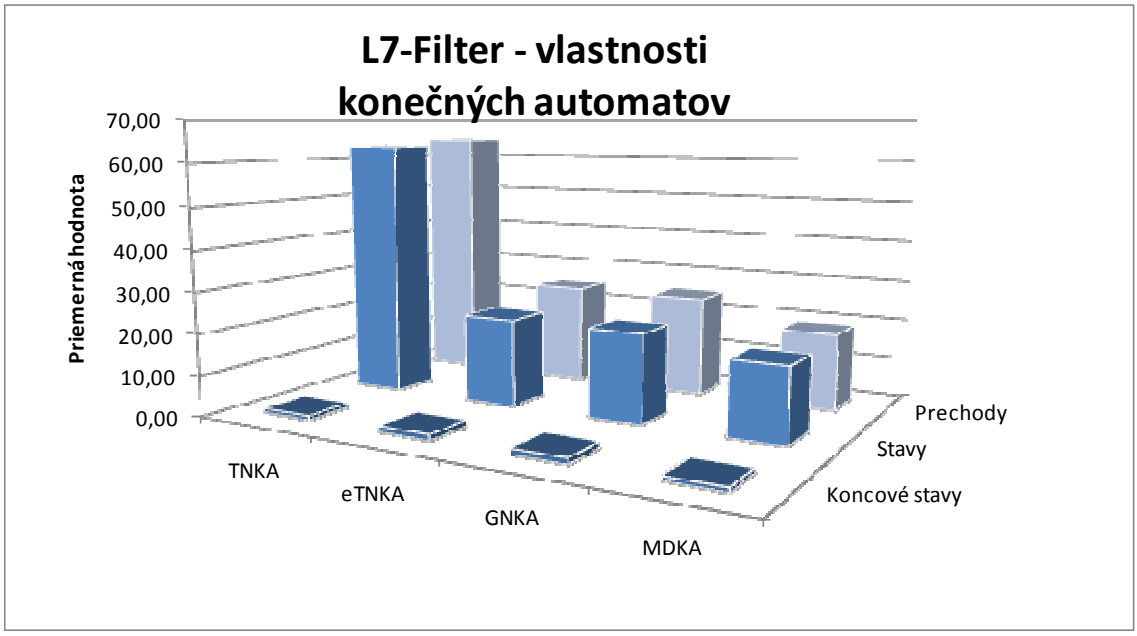
Pravidlá pre všetky aplikácie sa nachádzali vo viacerých súboroch, preto sme ich spojili do jedného súboru pre každú aplikáciu a predali ich nášmu programu ich dávkovo spracoval a vytvoril CSV súbor, ktorý sme importovali do tabuľkového procesoru a urobili štatistiky, resp. grafy.

5.3.1 L7-Filter



Z pravidiel L7-Filtera sa nám podarilo získať 278 jedinečných výrazov, všetky sa nám podarilo previesť na konečný automat. Každý regulárny výraz sa nachádzal v samostatnom súbore. Získané pravidlá boli vydané 10.5.2009.

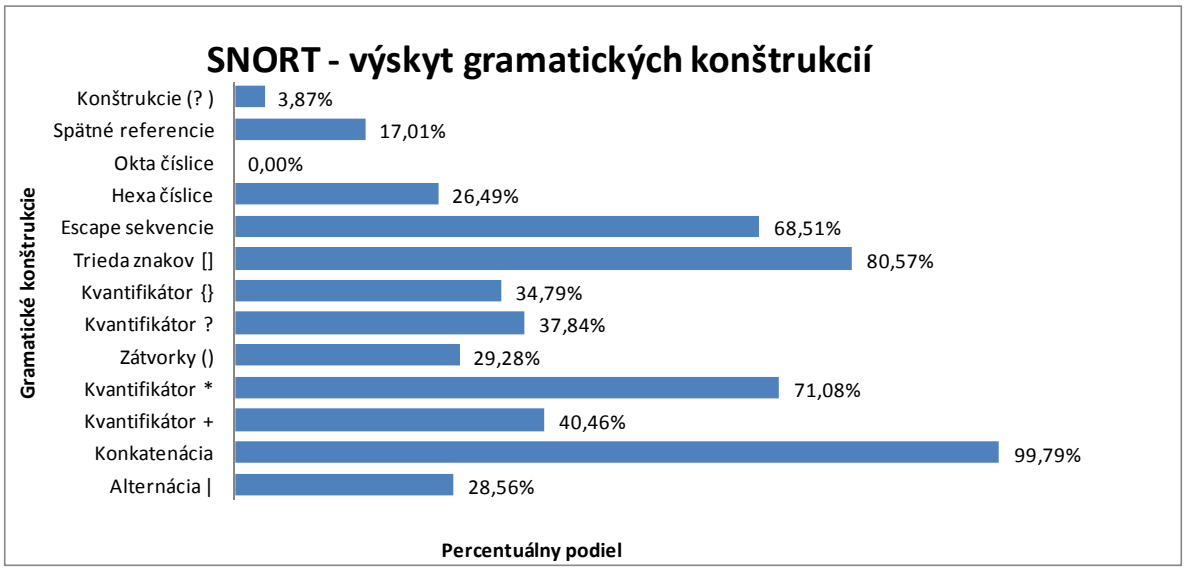
V pravidlách L7-filtera sme nenarazili na žiadne kvantifikátory {m,n}, {m} alebo {m,}. Taktiež sme nezaznamenali žiadne spätné referencie ani vyjadrenie znaku pomocou oktalovej hodnoty. Na druhej strane každý regulárny výraz obsahoval operáciu konkatenácie. Ostatné konštrukcie sa vyskytovali v menšej miere.



Vo výrazoch sa objavil jeden výraz, ktorý výrazne vybočoval a ovplyvňoval štatistiky počtom stavov a prechodov. Konkrétne sa jednalo o pravidlo „skypetoskype," z ktorého sme

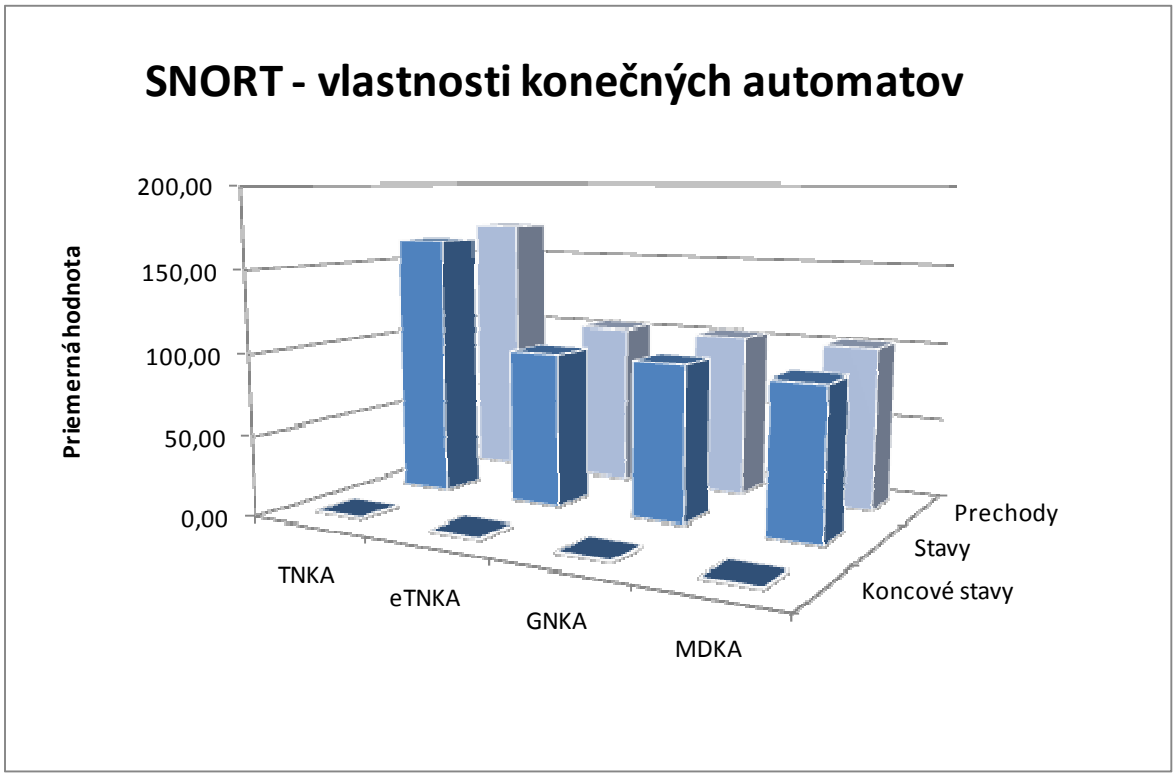
skonštruovali Thompsonov automat pozostávajúci z 11 732 stavov a 14 025 prechodov a Gluškovov automat z 2552 stavov a 11731 prechodov. Tento výraz sme sa rozhodli do štatistík nezapočítať.

5.3.2 SNORT

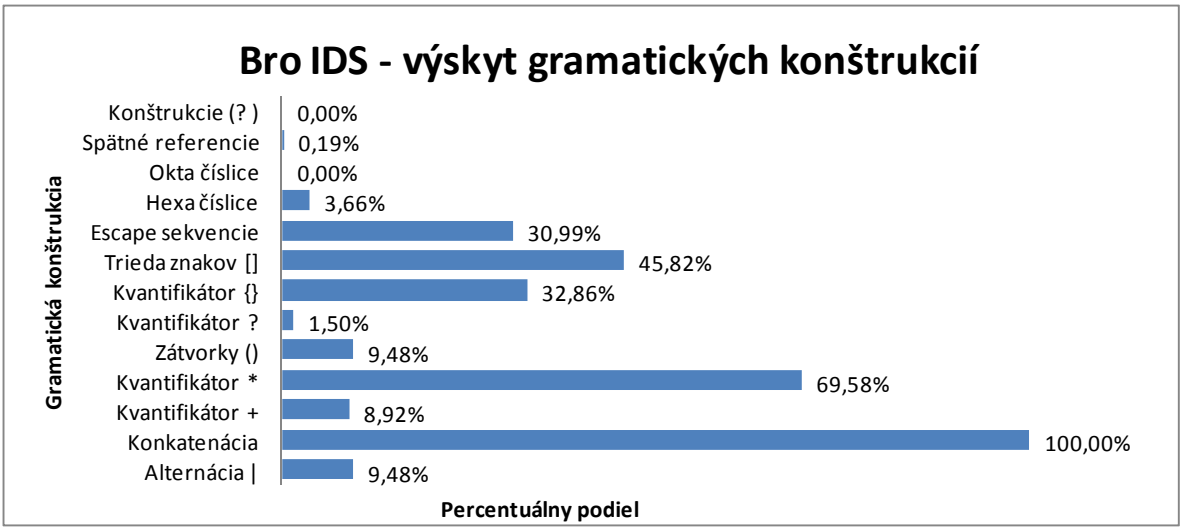


Aplikácia Snort nám ponúkla viacej regulárnych výrazov na spracovanie ako L7-Filter. Pravidlá boli vydané 14.4.2009. Získali sme približne 16340 výrazov, ale veľká väčšina bola duplicitná a po zredukovaní sme mali 1940 jedinečných výrazov, z toho z 1545 sa nám podarilo skonštruovať konečný automat. Relatívne veľkú vzorku sme museli ignorovať pre výskyt spätných referencií a v menšej miere nám toto číslo ovplyvnili taktiež konštrukcie *(?)*, ktoré sme nedokázali spracovať.

Z tejto vzorky výrazov obsahoval jeden spätné referencie. Dôležitým faktorom by mohol byť podiel použitia kvantifikátorov {m,n}, {m,}, {m}, ktoré obsahovalo až 34,79% výrazov. Za zmienku tiež stojí vysoký podiel tried znakov, ktoré sa nachádzali až v 80,57% výrazov a taktiež kvantifikátor * s podielom 71,08%. Vo všetkých výrazoch sa nachádzala operácia konkatenácie, na druhej strane neboli použité žiadne znaky reprezentované oktalovým zápisom.



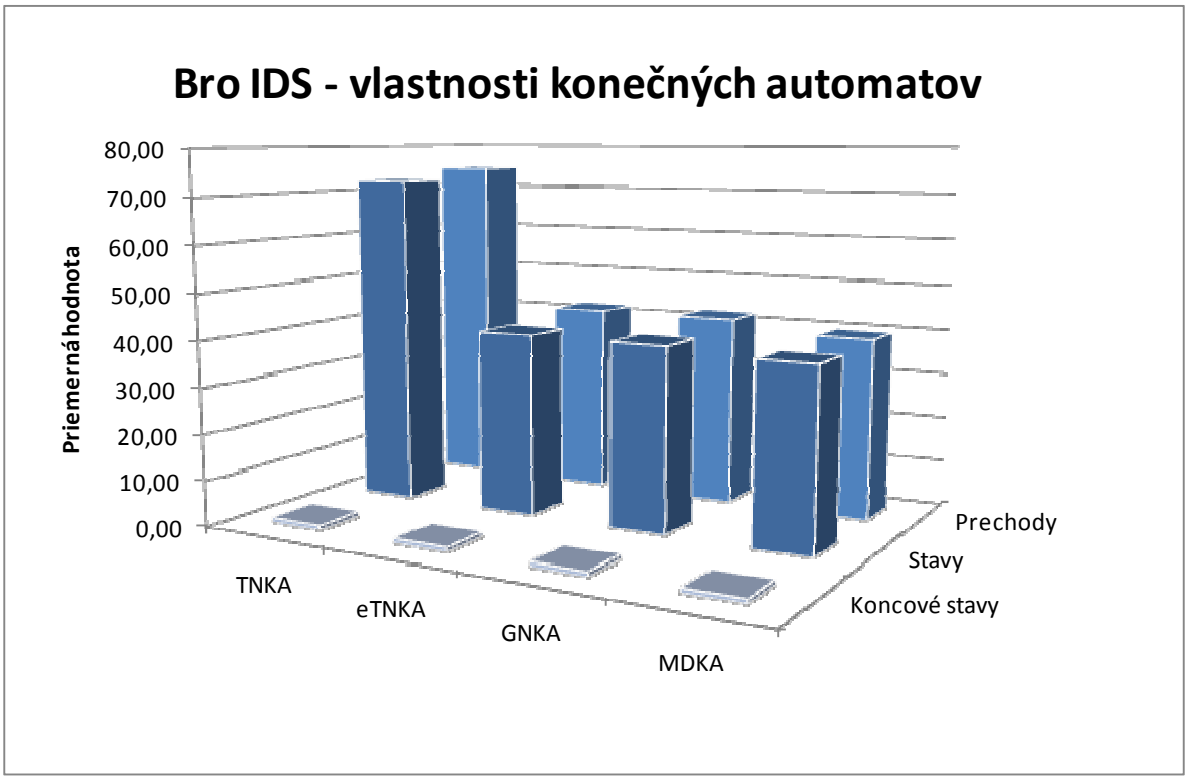
5.3.3 Bro Intrusion Detection System



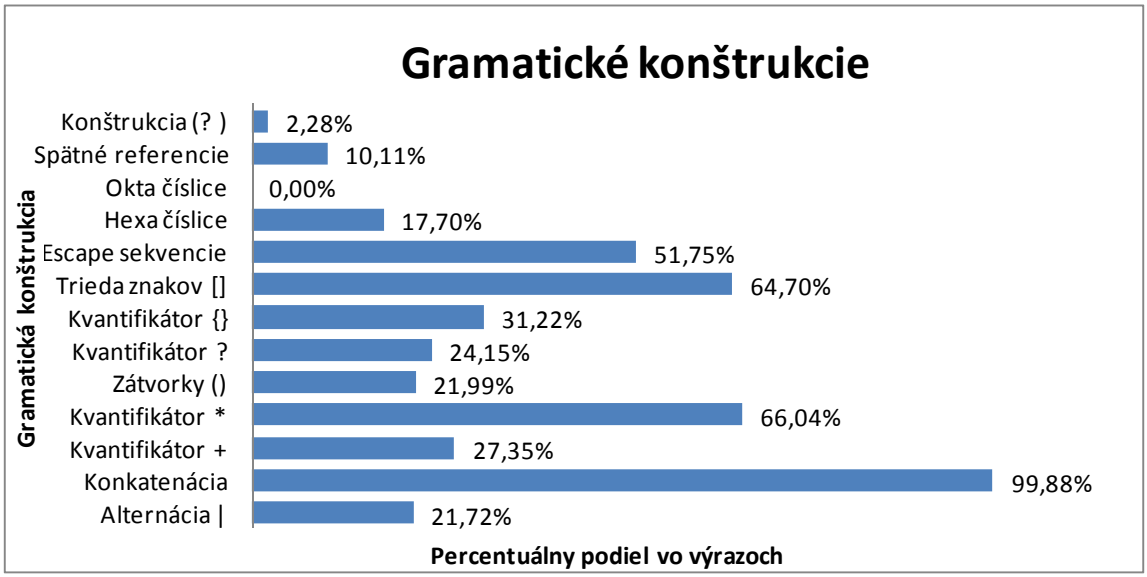
Aplikácia Bro IDS obsahuje vlastné pravidlá, ale využíva aj pravidlá z programu Snort. Použili sme pravidlá z verzie 1.4. Preto sme vybrali iba vlastné regulárne výrazy, ktoré používa Bro a pravidlá z aplikácie Snort sme ignorovali. Podarilo sa nám získať až 1065 jedinečných regulárnych výrazov, ktoré nám zostali po redukcii z celkového množstva 1587 výrazov, ktoré obsahovali aj duplicitné výrazy.

Bro IDS obsahoval relatívne vysoký podiel kvantifikátorov {m,n}, {m,}, {m}, až 32,86%. Nemenej dôležitý je tiež vysoký podiel kvantifikátoru * , ktorý dosiahol 69,58%. Takmer všetky výrazy využívali konkatenáciu a žiadny nevyjadroval znaky pomocou oktalovej hodnoty. Dva výrazy

obsahovali spätné referencie, čo predstavuje zanedbateľné množstvo. Žiadny výraz neobsahoval konštrukcie *(?)*.



5.4 Celkové výsledky



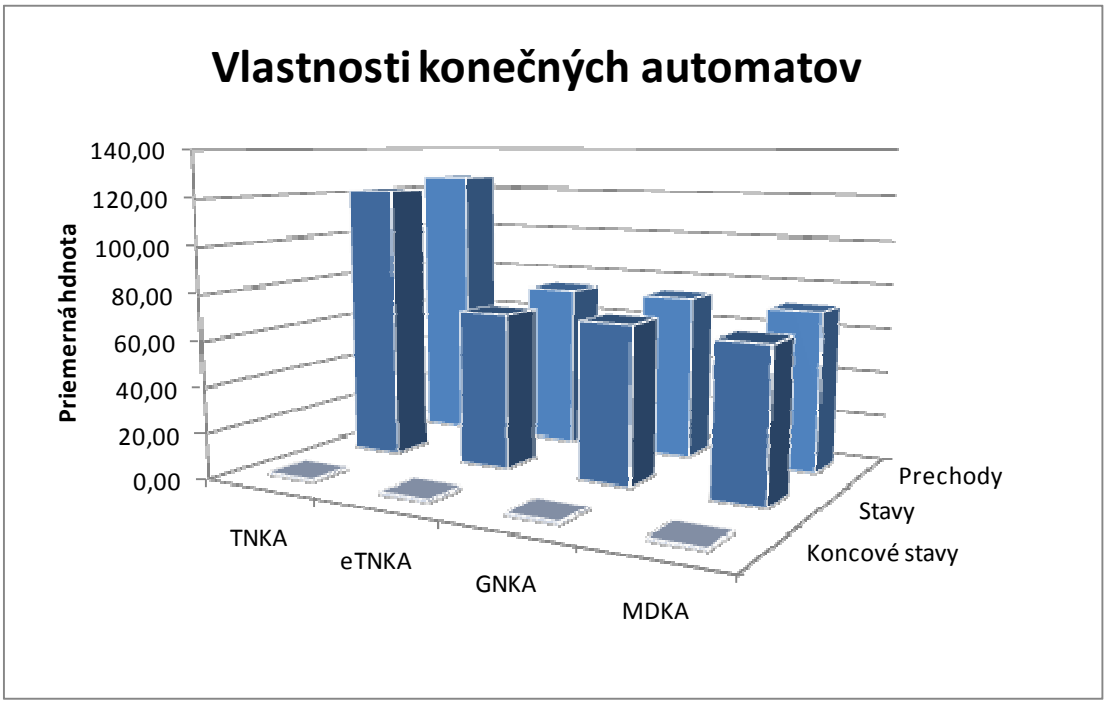
Celkovo sa nám zo všetkých testovaných aplikácií podarilo získať 3283 jedinečných výrazov. Najdôležitejšou gramatickou konštrukciou bola jednoznačne operácie konkatenácie, ktorú obsahovalo najväčšie množstvo výrazov, 99,70%. Relatívne vysoký podiel 25,85% získal taktiež kvantifikátor {m,n}, {m,} a {m}, keďže implementácia jeho konštrukcie je najzložitejšia zo všetkých gramatických

konštrukcií. Treba spomenúť, že tento kvantifikátor neobsahovala množina pravidiel L7-Filtera, teda pri implementácii by sme nemuseli tento kvantifikátor uvažovať.

Ani jeden regulárny výraz nepoužil vyjadrenie znaku pomocou oktalovej hodnoty, namiesto neho bola všade použitá hexadecimálna hodnota. Oveľa obľúbenejším spôsobom bolo vyjadrenie prijímaného znaku pomocou triedy znakov.

Zaujímavý je podiel konštrukcií *(?)*, z ktorých náš program nedokázal skonštruovať konečné automaty. Podiel 2,28% nie je pre celkové štatistiky zásadný.

Spätné referencie obsahovalo 332 čo znamená nezanedbateľný podiel 10.11%. Nakoľko je implementácia spätných referencií komplikovaná, tak je to relatívne vysoké číslo dôležitým faktorom pri implementovaní konštrukcie regulárnych výrazov.



	Koncové stavy	Stavy	Prechody
TNKA	**1,00**	**121,37**	**125,25**
eTNKA	**1,16**	**68,75**	**73,35**
GNKA	**1,16**	**68,75**	**73,35**
MDKA	**1,03**	**65,95**	**71,26**

Odstránením epsilon prechodov z Thompsonovho konečného automatu sme zredukovali počet stavov a taktiež počet prechodov takmer na polovicu. Túto redukciu sme realizovali pomocou FSM library a nezaznamenali sme žiadne citeľné spomalenie algoritmu, rádovo v sekundách pri najzložitejších výrazoch.

Môžeme si všimnúť, že všetky konečné automaty obsahovali veľmi málo koncových stavov. Thompsonov konečný automat dokonca obsahoval vždy iba jeden koncový stav, čo vyplýva aj z jeho definície konštrukcie, kedy má každý uzol stromu práve jeden začiatočný a jeden koncový stav.

Každý Gluškovov konečný automat sme zdeterminizovali a následne zminimalizovali pomocou FSM library. Výsledkom bol konečný automat s parametrami len o málo menšími ako pôvodný konečný automat. Toto platilo aj jednotlivo pre všetky výrazy.

Ďalej si môžeme všimnúť, že priemerné počty stavov a prechodov Gluškovovho konečného automatu a Thompsonovho konečného automatu sa vôbec nelíšia. V skutočnosti boli tieto hodnoty rovnaké pre všetky jednotlivé regulárne výrazy. Môžeme povedať, že pri testovaných výrazoch nám vždy z Thompsonovho konečného automatu po odstránení epsilon prechodov a prebytočných stavov vznikol konečný automat s rovnakými parametrami ako konečný automat zostrojený Gluškovovým algoritmom. Túto vlastnosť sa pokúsime vysvetliť nasledovne:

Vysvetlenie je založené na postupe odstránenia epsilon prechodov v kapitole 3.1. Pri konštrukcii Thompsonovho automatu nám vznikajú stavy do ktorých, resp. z ktorých vedú aj iné ako epsilon prechody. Tieto stavy nám vznikajú iba pri spracovaní jednotlivých znakov (prípadne kopírovaním konečného automatu pri aplikovaní kvantifikátorov {}), tzn. že nám vyjadrujú jednotlivé znaky regulárneho výrazu, podobne ako pri Gluškovovom prístupe. Podľa algoritmu odstránenia epsilon prechodov budú stavy, z ktorých, resp. do ktorých smerovali iba epsilon prechody odstránené, takže nám zostanú iba stavy reprezentujúce samotné znaky regulárneho výrazu, rovnako ako pri Gluškovovom prístupe. Vytvoria sa nám prechody medzi týmito stavmi. Epsilon prechody pri Thompsonovom prístupe slúžia iba k tomu, aby sme nemuseli riešiť problematiku viacerých koncových a začiatočných stavov alebo prijatie konečného automatu bez prečítania žiadneho znaku, tzv. epsilon stav pri Gluškovovom prístupe.

6 Záver

Opísali sme si gramatiku PCRE výrazov a vysvetlili sme si dva algoritmy konštrukcie nedeterministických konečných automatov z regulárnych výrazov. Výsledky z oboch algoritmov sme porovnali.

Na základe štatistík sme dospeli k záveru, že konštrukciou Gluškovovho a Thompsonovho konečného automatu získame dva zhodné automaty. Rozdiel medzi nimi je v ich konštrukcii, ale nemôžeme s určitosťou povedať, ktorý je vhodnejší.

Gluškovov konečný automat má výhodu v tom, že po jeho zostrojení ho nemusíme zbavovať epsilon prechodov. Thompsonov konečný automat využíva epsilon prechody, no na druhej strane nemusíme pri ňom uvažovať implementáciu „epsilon-stavov.“ Subjektívne bola jednoduchšia implementácia Thompsonovho konečného automatu, lebo odstránenie epsilon prechodov za nás vyriešila FSM library. Tá je však voľne dostupná iba pre nekomerčné účely. Pri komerčnom použití by bolo vhodnejšie zvoliť implementáciu pomocou Gluškovovho algoritmu.

Najjednoduchšie spracovávanou vzorkou výrazov boli pravidlá L7-Filtera, ktoré neobsahovali zložitejšie operácie a taktiež obsahovali málo pravidiel v porovnaní s ostatnými systémami. Najväčšiu vzorku PCRE výrazov sme získali z pravidiel systému Snort, ale na druhej strane obsahovali veľmi veľa spätných referencií, ktoré sme nedokázali previesť na konečný automat. Pravidlá Snortu boli významné tiež tým, že ich využívala aj aplikácia Bro IDS, a preto boli asi najzaujímavejšou vzorkou na spracovanie. Najzložitejšie konečné automaty sme generovali taktiež zo Snortu, počet stavov a prechodov bol asi dvakrát väčší ako pri ostatných IDS systémoch.

Táto práca nám pomohla lepšie pochopiť problematiku PCRE výrazov v IDS systémoch. Naučili sme sa vytvoriť precedenčnú tabuľku pre konkrétne pravidlá a vytvoriť z nich konečné automaty.

Námetom na ďalší výskum by mohol byť algoritmus na redukciu stavov a prechodov nedeterministických konečných automatov. Ďalej by mohla byť vytvorená voľne šíriteľná knižnica, ktorá by dokázala robiť rôzne operácie nad konečnými automatmi, tak ako FSM library, keďže je voľne použiteľná len pre nekomerčné účely.

Literatúra

[1] *Language reference - perle* [online]. 2006. Dostupný z WWW: <http://perldoc.perl.org/perlre.html>.

[2] GOYVAERTS, Jan. *POSIX Bracket Expressions* [online]. 2003-2009. Dostupný z WWW: <http://www.regular-expressions.info/posixbrackets.html>.

[3] *PCRE Pattern Syntax* [online]. 2009. Dostupný z WWW: <http://www.gambasdoc.org/help/doc/pcre>.

[4] HAZEL, Philip. *Pcresyntax man page* [online]. 2007. Dostupný z WWW: <http://regexkit.sourceforge.net/Documentation/pcre/pcresyntax.html>.

[5] COX, Russ. *Regular Expression Matching Can Be Simple And Fast* [online]. 2007. Dostupný z WWW: <http://swtch.com/~rsc/regexp/regexp1.html>.

[6] NAVARRO, Gonzalo, RAFFINOT, Mathieu. *Flexible pattern matching in strings*.: Cambridge University Press, 2002. 221 s. Dostupný z WWW: <http://books.google.com/books?id=GTTPGJctsMAC>. ISBN 0521813077.

[7] MEDUNA, Alexander, LUKÁŠ, Roman. *Formální jazyky a překladače - studijní opora*. : [2006]. 152 s.

[8] *Application Layer Packet Classifier for Linux* [online]. 2009. Dostupný z WWW: <http://l7-filter.sourceforge.net/>.

[9] *AT&T FSM Library – Finite-State Machine Library* [online]. Dostupný z WWW: <http://www.research.att.com/~fsmtools/fsm/>.

[10] *Bro Intrusion Detection System* [online]. 2003-2008. Dostupný z WWW: <http://www.bro-ids.org/>.

[11] *SNORT* [online]. 2009. Dostupný z WWW: <http://www.snort.org/>.

Zoznam príloh

Príloha 1. DVD.

- Zdrojové súbory programu
- Text práce v zdrojovom formáte
- Programová dokumentácia