VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

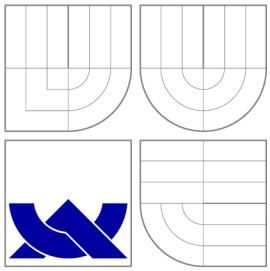
SROVNÁNÍ RYCHLOSTI MODERNÍCH SYSTÉMŮ PRO VYHLEDÁVÁNÍ REGULÁRNÍCH VÝRAZŮ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

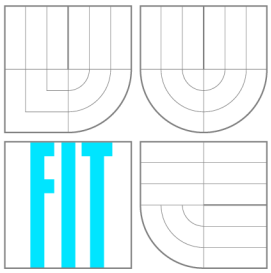
AUTOR PRÁCE
AUTHOR

JAN TRÁVNÍČEK

BRNO 2011



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SROVNÁNÍ RYCHLOSTI MODERNÍCH SYSTÉMŮ PRO VYHLEDÁVÁNÍ REGULÁRNÍCH VÝRAZŮ

COMAPRISSON OF THE SPEED OF THE MODERN SYSTEMS FOR REGULAR EXPRESSION MATCHING

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — JAN TRÁVNÍČEK
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN KAŠTIL
SUPERVISOR

BRNO 2011

Abstrakt

Tato bakalářská práce popisuje způsob srovnání rychlosti moderních nástrojů pro vyhledávání regulárních výrazů. Pro srovnání rychlosti jednotlivých nástrojů je použita množina regulárních výrazů z IDS systému Snort, kde jsou zadány v PCRE notaci. Tyto regulární výrazy jsou vyhodnocovány různými nástroji a získané výsledky jsou porovnávány mezi sebou. V této práci je také řešen matematický a praktický pohled na pojem regulární výraz a převod regulárních výrazů jazyka Perl do notace regulárních výrazů POSIX.

Abstract

This thesis describes how to compare the speed of modern tools for regular expressions matching. To compare the speed of each tool is used set of regular expressions from the Snort – Intrusion Detection System, which are specified in the PCRE notation. These regular expressions are evaluated by different tools and the results are compared with each other. In this work is also solved difference between mathematical and practical perspective on the term of regular expression and transfer Perl regular expressions in POSIX regular expressions.

Klíčová slova

Regulární výrazy, konečné automaty, PCRE knihovna, systém pro odhalení průniku, měření doby běhu aplikace

Keywords

Regular Expressions, finite automata, PCRE library, Intrusion Detection System, running time measurement applications

Citace

Jan Trávníček: Srovnání rychlosti moderních systémů pro vyhledávání regulárních výrazů, bakalářská práce, Brno, FIT VUT v Brně, 2011

Srovnání rychlosti moderních systémů pro vyhledávání regulárních výrazů

Prohlášení

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

.....
Jan Trávníček
18. května 2011

Poděkování

Rád bych tímto poděkoval svému vedoucímu bakalářské práce panu Ing. Janu Kaštilovi za odbornou pomoc, čas stráveny při konzultacích a za nezměrnou trpělivost, které se mi dostalo při řešení problémů.

Obsah

Kapitola 1

Úvod

Použití regulárních výrazů v dnešní době je velmi rozsáhle. Umožňují nám rychle a snadno vyhledat potřebné řetězce v textu. Nejčastěji se s regulárními výrazy setkáme v textových editorech, v programovacích a skriptovacích jazycích nebo při kontrole síťového provozu. Právě sytémy pro detekci a filtrování síťových paketů mají velkou náročnost na rychlost vyhodnocování regulárních výrazů. Požaduje se co možná nejmenší doba vyhodnocení paketů, čili i samotných regulárních výrazů. Proto má smysl srovat nástroje a vybrat si pro tento účel ten nejrychlejší a nejvhodnější.

Nejznámější IDS systémy jsou aplikace Snort [24] a Bro. V této práci se zabýváme regulárními výrazy, které používá systém Snort. Tento systém používá pro vyhodnocování regulárních výrazů knihovnu PCRE [10], proto je výchozí množina regulárních výrazů v PCRE notaci. V této práci porovnáváme rychlost knihovny PCRE s několika dalšími nástroji: Perl, GNU grep, GNU sed, GNU awk a mawk. Tyto nástroje, kromě jazyka Perl, podporují regulární výrazy, jak je definuje standard POSIX. Z toho důvodu je prováděn převod mezi různými standardy regulárních výrazů.

Druhá kapitola obsahuje popis základních pojmů jako regulární jazyk, regulární výraz a konečný automat z matematického hlediska. Třetí kapitola popisuje regulární výrazy, které se prakticky používají v dnešní době. Převod z notace PCRE do dalších skupin regulárních výrazů je popsán v kapitole čtyři. Kapitola pět popisuje implementaci a testované nástroje. V šesté kapitole rozebíráme nejhorší případy pro různé přístupy. Naměřené výsledky jsou zobrazeny a diskutovány v kapitole sedm.

Kapitola 2

Úvod do regulárních výrazů

Tato kapitola obsahuje shrnutí několik základních pojmů. Obsahuje popis pojmu regulárního jazyka, regulárních výrazů a konečných automatů. Dále se v této kapitole popisuje rozdíl a základní princip konečných automatů používaných pro vyhodnocování regulárních jazyků.

2.1 Regulární jazyky a výrazy

V této kapitole jsou popsány základní pojmy jako regulární jazyk, regulární výraz a regulární operace.

2.1.1 Regulární jazyky

Regulární jazyky [3] jsou definovány následujícím způsobem. Třída $RJ\{\Sigma\}$ regulárních jazyků nad abecedou Σ je nejmenší třída jazyků nad Σ splňující tyto podmínky:

1. $\emptyset \in RJ(\Sigma)$ a $\{a\} \in RJ(\Sigma)$ pro každé $a \in \Sigma$
2. $L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1 \cup L_2 \in RJ(\Sigma)$
3. $L_1, L_2 \in RJ(\Sigma) \Rightarrow L_1.L_2 \in RJ(\Sigma)$
4. $L \in RJ(\Sigma) \Rightarrow L^* \in RJ(\Sigma)$

Operace sjednocení, konkatenace a iterace se někdy nazývají *regulární operace nad jazyky.* Lze tedy také říci, že regulární jazyky jsou právě jazyky, které lze dostat z elementárních jazyků aplikací konečně mnoha regulárních operací. Elementárními jazyky zde rozumíme prazdný jazyk a jazyky tvořené jediným slovem o jednom symbolu.

Z definice iterace 2.1.1 je patrné, že i jazyk obsahující pouze prázdné slovo je regulární, neboť $\{e\} = \emptyset^*$ [3].

Definice 2.1.1 *Iteraci L^* jazyka L definujeme jako:*

$$L^* = L^0 \cup L \cup L^2 \cup L^3 \cup \cdots = \bigcup_{i=0}^{\infty} L^i$$

2.1.2 Regulární výrazy

Každý regulární jazyk je zřejmě možné zadat stanovením příslušných elementárních jazyků a předpisů, jak na ně aplikovat regulární operace. K tomuto účeky slouží tzv. regulární výrazy [3]. Množinu $RV(\Sigma)$ regulárních výrazů nad abecedou $\Sigma = \{a_1, \ldots, a_n\}$ definujeme jako nejmenší množinu slov v abecedě $\{a_1, \ldots, a_n, \emptyset, e, +, ., ^*, (,)\}$, kde e, $\emptyset$, $+$, $.$, *, $($ a $)$ jsou symboly nepatřící do abecedy Σ splňující tyto podmínky:

1. $\emptyset \in RV(\Sigma)$, $e \in RV(\Sigma)$, $a \in RV(\Sigma)$ pro každé $a \in \Sigma$
2. $\alpha, \beta \in RV(\Sigma) \Rightarrow (\alpha + \beta) \in RV(\Sigma), (\alpha.\beta) \in RV(\Sigma), \alpha^* \in RV(\Sigma)$

Každý z regulárních výrazů označuje jistý regulární jazyk. Výraz $\emptyset$ označuje prázný jazyk, e označuje jazyk $\{e\}$, pro $a \in \Sigma$ označuje a jazyk $\{a\}$. Jestliže α, β jsou výrazy označující po řadě jazyky L_1, L_2, potom $(\alpha + \beta)$ označuje $L_1 \cup L_2$, $(\alpha.\beta)$ označuje $L_1.L_2$ a a^* označuje L_1^*. Obecně budeme jazyk reprezentovaný regulárním výrazem α označovat $[\alpha]$.

Pro zpřehlednění zápisu regulárních výrazů se zpravidla vynechávají některé závorky. Vynecháváme vnější pár závorek, dále můžeme vypustit některé závorky díky tomu, že operace součinu a sjednocení jazyků jsou operace asociativní. Další závorky lze vynechat například díky prioritě jednotlivých regulárních operací. Nejvyšší prioritu má operace * a nejnižší naopak operace $+$.

2.2 Konečné automaty

Konečný automat *(Finite Automata)* [3] je obecně definován následovně:

Definice 2.2.1 *Konečným automatem nazýváme každou pětici $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$, kde Q je konečná neprázdná množina (množina stavů, stavový prostor), Σ je konečná neprázdná množina (množina vstupních symbolů, vstupní abeceda), δ je zobrazení $Q \times \Sigma \to Q$ (přechodová funkce), $q_0 \in Q$ (počáteční stav, iniciální stav), $F \subseteq Q$ (cílová množina, množina koncových stavů).*

Pokuď definujeme konečný automat, musíme zadat všech pět položek. Konečný automat lze zapsat i jinými, ekvivalentními způsoby – přechodovým diagramem nebo tabulkou [26].

2.2.1 Deterministický konečný automat – DFA

Deterministický konečný automat *(Deterministic Finite Automaton)* [26] je pětice

$$M = (Q, \Sigma, \delta, q_0, F)$$

, kde

- Q je konečná množina; prvky Q jsou stavy.
- Σ je konečná množina; vstupní abeceda.
- $\delta : Q \times \Sigma \to Q$ je přechodová funkce (jinými slovy $Q \times \Sigma$ je množina ospořádaných dvojic $\{(q, a) | q \in Q \text{ and } a \in \Sigma\}$). Takže δ je přechodová funkce, která nám říká jaký stav bude výsledkem pro daný vstup: když M je ve stavu q a vstupem je a, přejde se do stavu $\delta(q, a)$.
- $q_0 \in Q$ je počáteční stav.
- F je podmnožinou Q; prvky F jsou nazývány koncovými stavy.

2.2.2 Jazyky DFA

Nechť $a_1, a_2, \ldots, a_n$ je posloupnost vstupních symbolů. Vyhodnocování začíná tím, že DFA je v počátečním stavu, q_0. Nahlédneme do přechodové funkce δ, která nám řekne následující stav $\delta(q_0, a_1) = q_1$, do kterého přejde DFA M po přijetí vstupního symbolu a_1. Po té se vyhodnotí další vstupní symbol, a_2, nahlednutím do přechodové funkce $\delta(q_1, a_2) = q_2$. Přechodová funkce nám vrátí další stav, q_2, do kterého se přejde. Stejným způsobem se pokračuje pro nalezení dalších stavů $q_3, q_4, \ldots, q_n$. Tento postup můžeme obecněji zapsat: $\delta(q_{i-1}, a_i) = q_i$ pro každé i. Pokuď výsledný stav q_n je členem množiny F, potom vstupní posloupnoust symbolů $a_1, a_2, \ldots, a_n$ je přijata. V opačném případě je odmítnuta [13].

2.2.3 Nedeterministický konečný automat – NFA

Nedeterministický konečný automat *(Nondeterministic Finite Automaton)* [26], je takový, kde další stav není jednoznačně dán tím, v jakém stavu se takový automat nachází společně se vstupním symbolem. V deterministickém konečném automatu je přesně jeden počáteční stav a přesně jeden přechod pro každý stav a pro každý symbol z Σ. Z toho plyne, že je vždy pouze jeden aktivní stav. V nedeterministickém automatu může být jeden nebo víc než jeden počáteční stav. Obdobně může NFA mít jeden nebo více aktivních stavů. Bavíme se o množině možných stavů, do kterých se automat může posunout ze stavu q po přijetí vstupního symbolu a. Neexistuje ale žádný mechanismum k určení, který z těchto stavů má být použit. Nedeterministický konečný automat má také obvykle mnoho počátečních stavů a samotný start může být proveden z jakéhokoliv.

Nedeterministický konečný automat je tedy definován jako pětice

$$N = (Q, \Sigma, \Delta, S, F)$$

, kde Q. Σ a F jsou stejné jako u DFA. Další použité znaky mají následující význam.

- S je množina stavů, kde $S \subseteq Q$. Prvky S jsou nazývány počáteční stavy.
- Δ je funkce $\Delta : Q \times \Sigma \to 2^Q$, kde 2^Q značí množinu všech podmnožin Q: $2^Q = \{A | A \subseteq Q\}$.

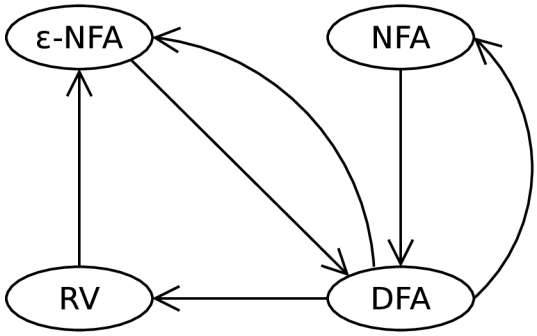
2.2.4 ϵ-NFA

ϵ-NFA je takový konečný automat, který obsahuje epsilon přechody. Epsilon přechod dovoluje konečnému automatu přejít spontáně z jednoho stavu do jiného bez přijetí vstupního symbolu. Nezvětšuje tím množinu jazyků, které takový konečný automat může přijímat. Epsilon přechody nám pouze umožňují přehlednější a jiný zápis, neboť vidíme lépe souvislosti s regulárními výrazy [13].

2.3 Ekvivalence konečných automatů a regulárních výrazů

Je dokázáno, že každý regulární jazyk je rozpoznatelný konečným automatem. A stejně tak i naopak, každý jazyk rozpoznatelný konečným automatem je regulární. Z tohoto tvrzení nám vyplývá, že můžeme daný jazyk zapsat jakýmkoliv z těchto způsobů [3]. Je i jedno, kterým konečným automatem je takový jazyk zapsán. Víme, že DFA, NFA i ϵ-NFA definují stejnou třídu jazyků. Mezi těmito čtyřmi způsoby zápisu můžeme tedy vždy určitým způsobem docílit zápisu pomocí jiného. Na obrázku 2.1 vidíme vztah mezi čtyřmi ekvivalentními

zápisy pro regulární jazyky [13]. Z obrázku můžeme vidět, že z každého uzlu se můžeme dostat do jakéhokoliv jiného uzlu. I z toho si můžeme povšimnout ekvivalence mezi různými zápisy.



Obrázek 2.1: Ekvivalence mezi čtyřmi možnými zápisy pro regulární jazyky.

Kapitola 3

Prakticky používané regulární výrazy

Tato kapitola popisuje regulární výrazy, jak je známe, chápeme a používáme v dnešní době v běžně používaných nástrojích. V minulé kapitole jsme si popsali regulární jazyky a regulární výrazy z matematického hlediska. V dnešní době už se ale pojem regulární výraz nepoužívá zcela přesně.

V praxi byla snaha zvětšovat sílu regulárních výrazů a nabídnout tak uživatelům nástroje z větší sílou. Dělo se tak například pomocí přidání backreferencí. Prodloužila se také ale doba potřebná pro vyhledávání, protože vyhledání backrefernce již není možné provést v linearním čase. Pokud backreferenci nevyužijeme, vetšina nástrojů používá přesto stejný vyhledávací algoritmus, který není tak efektivní jako konečné automaty, a to vede ke zvyšovaní složitosti i pro výrazy bez backreferencí. Nástroje, které podporují backrefenci, již přijímaji neregulární jazyky. Z toho důvodu je nutné rozlišovat regulární výraz z pohledu matematika a regulární výraz z pohledu uživatele těchto nástrojů. V běžné praxi se toto odlišení provádí pojmem „RegExp“, který není významově shodný s regulárními výrazy [6].

Dále jsou v této kapitole popsány rozdílné implementace NFA, s kterými se můžeme setkat v praxi. Podle literatury [6] je předveden a vysvětlen princip jednoduchého testu pro rozpoznání algoritmu nástroje.

3.1 Regulární výrazy POSIX

Podle standardu POSIX [22] – Portable Operating System Interface – jsou regulární výrazy rozděleny do dvou skupin [6]. Jedna skupina se označuje jako „Basic Regular Expressions“, zkratka BRE, a druhá jako „Extended Regular Expressions“, zkratka ERE. Do češtiny bychom tyto dvě skupiny mohli přeložit jako základní regulární výrazy a rozšířené regulární výrazy. V tabulce č. 3.1 můžeme vidět, jakou skupinu regulárních výrazů používá daný nástroj.

3.1.1 Základní regulární výrazy – BRE

Základní regulární výrazy BRE [17] používá znak `.` (tečka), kterému odpovídá jakýkoliv znak kromě znaku nového řádku `\n`. Pokud chceme, aby nám znak `.` odpovídal i znaku nového řádku `\n`, musí se explicitně nástroj přepnout do tzv. režimu *dot-match-all* někdy také označovaného jako *single-line*. Tato skutečnost je zachována z historických důvodů, když se

Nástroj	Skupina regulárních výrazů
Perl	Perl regulární výrazy
PCRE	Perl regulární výrazy
sed	BRE + GNU rozšíření
sed -r	ERE + GNU rozšíření
grep	BRE + GNU rozšíření
egrep	ERE + GNU rozšíření
GNU awk	ERE + GNU rozšíření
mawk	ERE + GNU rozšíření

Tabulka 3.1: Podporované regulární výrazy v jednotlivých nástrojích.

ujalo pravidlo, že výráz `.*` odpovídá zbytku řádku až po znak nového řádku `\n`. Toto pravidlo vyplývá z nástrojů jako *grep* a *sed*, které pracovaly výhradně s jedním řádkem řetězce, nad kterým se vyhledávalo. Dále definuje znaky `^` pro začátek řádku a `$` pro konec řádku. Množina znaků se zadává běžným způsobem `[...]` a odbobně negovaná množina `[^...]`. Jako jediný kvantifikátor je zde používán znak hvězdička `*`, který znamená vyskyt nula a vícekrát. Interval výskytu se zadává pomocí znaků `\{min,max\}`. Pokud je zadána pouze jedna hodnota, jedná se o výskyt přesného množství. Další možné zápisy jsou `\{min,\}`, což znamená výskyt *min*-krát a vícekrát a obdobně pro `\{,max\}` výskyt nula až *max*-krát. Pro vytváření skupin se používá zápis `\(...\)`. Pro zápis backreference je použit zápis `\m`, kde je m je číslo v rozsahu 1 až 9.

3.1.2 Rozšířené regulární výrazy – ERE

Rozšířené regulární výrazy ERE [18] definují metaznaky `.`, `^`, `$`, `[...]`, `[^...]` se stejným významem jako BRE. Odlišnosti v zápisu se projevují v následujících metaznacích. Složené závorky uvozující interval není již třeba escapovat [12] `{min,max}`. Obdobně i závorky pro skupiny `(...)`. Význám escapování závorek je tedy přesně opačný než u BRE.

Dále ERE definuje nové metaznaky – `+`,`?` a `|`. Metaznak `+` je kvantifikátor, který znamená výskyt jednou a vícekrát. Otazník jako metaznak představuje také kvantifikátor, který má význam výskyt nula nebo jedenkrát. Nejdůležitějším metaznakem, který byl nově v ERE definován, je znak roura `|`, který představuje alternaci. Výraz „`ab|cd`“ tedy znamená, že tomuto výrazu bude odpovídat jak řetezec „ab“, tak řetězec „cd“.

V ERE nebyla zakomponována podpora pro backreference. Ty byly přidány až GNU rozšíření [18].

3.1.3 GNU rozšíření

Pro zjednodušení regulárních výrazů bylo vydáno GNU rozšíření pro obě skupiny. Hlavní snahou bylo přidat podporu toho, co jeden nástroj podporuje a druhý ne. Uživatel si tak nemusí pamatovat, co která skupina přesně podporuje a co ne. Jediným rozdílem tak zůstává pouze escapování některých metaznaků v BRE a v ERE je význam escapování přesně opačný.

Do BRE byly přidány tedy další tři metaznaky – `\+`,`\?` a `\|`. Tyto znaky mají stejný význam jako jejich obdoba v ERE. Do ERE byla naopak přidána právě podpora backreference, jak ji známe z BRE [7].

V dnešní době se již velmi nerozlišuje zda tento metaznak je přímo ve standardu určité

skupiny nebo v GNU rozšíření. Je právě naopak běžné, že je v nástroji implementována i podpora pro GNU rozšíření.

3.1.4 Třídy znaků POSIX

Ve standardu POSIX jsou také definovány tzv. třídy znaků. Jejich výčet můžeme vidět v tabulce č. 3.2. Pokuď se podíváme například na třídu `[:lower:]`, mohli bychom ji přepsat pomocí množiny `[a-z]`. To ovšem není úplně korektní. V třídě `[:lower:]` jsou zahrnuty jak malá písmena, tak malá písmena s danou lokalizací, takže například písmena s čárkou nebo háčkem.

Třída	Odpovídající ASCII množina
`[:alnum:]`	`[a-zA-Z0-9]`
`[:alpha:]`	`[a-zA-Z]`
`[:blank:]`	`[ \t]`
`[:cntrl:]`	`[\x00-\x1F\x]7F`
`[:digit:]`	`[0-9]`
`[:graph:]`	`[\x21-\x7E]`
`[:lower:]`	`[a-z]`
`[:print:]`	`[\x20-\x7E]`
`[:punct:]`	`[][!"#$%&'()*+,./:;<=>?@^_‘{|}~-]`
`[:space:]`	`[ \r\n\t\v\f]`
`[:upper:]`	`[A-Z]`
`[:xdigit:]`	`[a-fA-F0-9]`

Tabulka 3.2: Třídy znaků POSIX.

3.2 Regulární výrazy jazyka Perl

Skriptovací jazyk Perl přinesl několik nových konceptů do regulárních výrazů, které dále přebíraly další jazyky jako např Python. V jazyku Perl jsou definovány metaznaky – `.`, `^`, `$`, `[...]`, `[^...]`, {`min,max`}, (...), `*`, +, `?`, `|`, `\m`. Tyto znaky mají stejný význam jako jejich obdoba ve skupinách regulárních výrazů POSIX.

Jazyk Perl definuje další metaznaky. Přibyla podpora pro metaznaky `\w`, `\s` a `\d` pro třídy znaků. Význam těchto metaznaků a jejich komplementů `\W`, `\S`, `\D` můžeme vidět v tabulce 3.3, kde jsou porovnávány s množinami znaků, které jim odpovídaji v ASCII. Toto ale není naprosto přesné, neboť třídy znaků v Perlu často zahrnují i znaky z abecedy s lokalizací. To znamená, že přijímají znaky včetně čárek, háčku a kroužků, pokuď se omezíme na českou lokalizaci.

Dále Perl zavádí modifikátor `/i`, který způsobí vyhodnocování regulárního výrazů jako *case-insensitive*, tedy necitlivý na rozdíl malých a velkých písmen. Další modifikátor, který Perl přidává, je `/x`, který nám dovoluje psát regulární výraz přehledněji s bílými mezerami a s komentáři. Tento přepínač má zpřehlednit a urychlit pochopení významu zapsaného regulárního výrazu. Důležité modifikátory, který byly definovány jazykem Perl, jsou `/s` (viz odstavec č. 3.1.1) a `/m`. Modifikátor `/m` značí tzv. *multi-line* mód. Tento mód způsobí, že kotvy `^` a `$` odpovídají každému začátku a konci řádku, pokuď se pracuje s více řádkovým řetězcem. Bez použití tohoto módu tyto kotvy odpovídají začátku a konci celého řetězce.

Metaznak	Odpovídající ASCII množina
\w	[a-zA-Z0-9_]
\W	[^a-zA-Z0-9_]
\d	[0-9]
\D	[^0-9]
\s	[\r\n\t\v\f]
\S	[^ \r\n\t\v\f]

Tabulka 3.3: Třídy znaků podle jazyka Perl.

Dále je také přidána skupina, která není brána v úvahu pro backreferenci – *non-capturing* skupina. Taková skupina se zapisuje (`?:...`). Na takovou skupinu se potom nelze odkazovat pomocí backreference. Nově je přidán také tzv. „líný kvantifikátor“, který je zapisován pomocí metaznaku otazník `?` za kvantifikátorem – např. `*?`, `+?` nebo `??`. Někdy je také označován jako „nehladový kvantifikátor“ (*ungreedy*). Jeho význam si vysvětleme na příkazu. Pokud aplikujeme výraz `<.+?>` na řetězec „`<span>text</span>pokracovani`“, bude vyhodnocen jako „`<span>`“. Pro hladový kvantifikátor by byl ale výsledek „`<span>text</span>`“.

Další vlastností regulárního výrazu, kterou zavedl jazyk Perl je tzv. nahlížení. Jedná se o nahlížení vpřed (`?=...`) a vzad (`?<=...`) a záporné nahlížení vpřed (`?!...`) a vzad (`?<!...`). Nahlížení zpět má omezení v tom, že musí být předem znám počet znaků, které budou odpovídat. Z toho plyne, že v nahlížení zpět nesmí být kvantifikátor nebo zadán interval opakování. Zápis (`:...`) uvozuje skupinu, která nebude brána v úvahu pro backreference nebo nebude součástí výsledku vyhodnocení regulárního výrazu. Označuje se názvem „Non-Capturing Group“.

Dále jazyk Perl zavedl atomické vyhodnocování skupiny, které je uvozeno metaznakem (`?>...`). Znamená to, že pokuď jednou začne vyhodnocovat určitou část takové skupiny, již se nesnaží při neúspěchu vyhovět jinou částí. Význam si objasníme na příkladu. Regulárnímu výrazu `a(bc|b)c` odpovídá jak řetězec `abcc` tak i řetězec `abc`. Pokud ale použijeme atomické vyhodnocování skupiny, regulárnímu výrazu `a(?>bc|b)c` odpovídá řetězec `abcc` ale již ne řetězec `abc`. Symbol `a` je přiřazen výrazu `a` před skupinou. Symbol `b` je přiřazen první možnosti v atomické skupině, tedy výrazu `bc`. Tomuto výrazu odpovídá i další symbol `c`. Po té ale již další symbol nenásleduj a tudíž je vyhledávání ukončeno s negativním výsledkem. Nástroj se nevrací k druhému výrazu ve skupine za alternací `b` a nesnaží se vyhodnotit další variantu [8].

3.2.1 Knihovna PCRE

Koncepci zapisování regulárních výrazů, jak jsou definovány jazykem Perl, převzalo později několik jazyků – např. Python, Java. Tato koncepce skriptovacího jazyka Perl se stala vedoucím a hlavním směrem, který se rozšířil dále a hlavně se ujal. Rozšíření napomohl vznik knihovny PCRE [10], která měla nabídnout Perl regulární výrazy přístupné snadno a všem, proto je dostupná jako opensource. Tato knihovna zachovává syntaxi a sémantiku, která je shodná z regulárními výrazy v Perlu. Nepatrné rozdíly existují [11], ale většinou vyplývají z rozdilné implementace a jejich nepatrný dopad většinou běžný uživatel ani nepostřehne. Je ale potřeba je brát v úvahu. Zde si uvedeme pouze stručný výčet známých rozdilů.

- Rekurzivivní vyhodnocování je v PCRE atomické, ale v Perl je neatomické.
  Více o atomickém vyhodnocování v odstavci o regulárních výrazech jazyka Perl 3.2.

- Způsob vyhodnocování `?` kvantifikátoru (`{0,1}`), pokuď je zanořen pod jiným kvantifikátorem.
- PCRE umožňuje, aby jméno pojmenovávané backreference bylo číslo.
- PCRE nepodporuje všechny nejnovější experimentální konstrukce, které nám nabízí Perl.
- PCRE a Perl se jemně odlišují v toleranci pro zápis chybných konstrukcí.
- PCRE má limit na hloubku rekurze, zatímco Perl ho nemá.

Knihovna PCRE je používáná několika velkými projekty jako jsou např. Apache, PHP, KDE, Postfix, Analog [10] a Snort [24].

3.2.2 PCRE výraz

Knihovna používá pro zápis regulárních výrazů svůj tvar zápisu – `/RV/modifikatory`. `RV` je regulární výraz, jak jej známe z popisu regulárních výrazů Perl. Část `modifikatory` jsou jednoznakové modifikátory, kterých úplný výčet můžeme najít v manualový stránkách [9]. V mé práci jsem bral v úvahu pouze ty nejpoužívanější modifikátory. Jejich výčet můžeme vidět v tabulce č. 3.4.

Modifikátor	Význam
`s`	Single-line mód
`m`	Mulit-line mód
`i`	Case-insensitive
`g`	Globální vyhledávání
`U`	Ungreedy vyhledávání

Tabulka 3.4: Podporované modifikátory v této práci.

V této formě testujeme PCRE regulární výrazy, které dostaneme z pravidel aplikace Snort. Na vyparsování PCRE výrazů z archivu těchto pravidel byl naimplementován skript, který zařídí parsování automaticky a PCRE výrazy nám uloží do potřebného formátu.

3.3 Rozdílné implementace NFA

V praxi se nejčastěji používají dvě různé implementace nedeterministických konečných automatů. Jedná se o tzv. Tradiční NFA a POSIX NFA. Jejich základní rozdíl si vysvětlíme na jednoduchém příkladu, jak je to vysvětleno i v literatuře [6]. Existují i další implementace, které ale nejsou příliš rozšířené, proto se jimi nebudeme zabývat.

Mějme regulární výraz `one(self)?(selfsufficient)?` a vyhodnocujme ho nad řetezcem `oneselfsufficient`. Implementace Tradiční NFA vyhodnotí jako výsledek `oneself`. Dalo by se říci, že se spokojí s první možnou variantou, která odpovídá regulárnímu výrazu.

U implementace POSIX NFA je to ale jiné. Tato implementace se drží pravidla *nejdelší zleva*. To znamená, že zde bude jako výsledke vyhodnocení vrácen celý řetězec `oneselfsufficient`.

Nejčastější implementace DFA, se kterou se můžeme setkat, se také drží pravidla *nejdelší zleva*. Proto se snaží právě implementace POSIX NFA chovat stejně jako DFA.

3.3.1 Použité algoritmy v nástrojích

V tabulce č. 3.5 vidíme zastoupení jednotlivých variant implementace v testovaných nástrojích. Nástroje byly vybírány tak, aby byla zastoupena každá skupina. Jednotlivé implementace se liší téměř nástroj od nástroju. Tabulka spíše zobrazuje algoritmus, na kterém je konkrétní implementace založena [6].

Typ algoritmu	Nástroj
Tradiční NFA	PCRE, Perl, GNU sed
POSIX NFA	mawk
Hybridní NFA/DFA	GNU awk, GNU grep

Tabulka 3.5: Použité algoritmy v testovaných nástrojích [6].

V dnešních moderních nástrojích se již implementace založené na DFA příliš nepoužívají. Můžeme se s ní setkat např. u původní implementace jazyka awk nebo u MySQL [6]. V dnešní době se místo této varianty používá kombinovaná metoda – Hybridní přístup. Tato variant má mnoho podob a variant, jak ji lze chápat a implementovat.

Například GNU grep se snaží použít DFA, kde je to vhodné a rychlejší, a naopak použít NFA, pokuď je třeba vyhodnocovat složité konstrukce. GNU awk funguje na jiném principu. Nejdříve se pokusí použít rychlý DFA aparát pro jednoduché vyhodnocení *zleva nejdelší* shody a po té se zpětně snaží kontrolovat, kde je třeba vyhodnotit složitější konstrukci. Pro tento druhý přístup je právě použit NFA algoritmus [6].

3.4 Jednoduchý test na rozpoznání algoritmu

Tento odstavec obsahuje popis jednoduchého testu, kterým můžeme zjistit, který algoritmus používá daný nástroj. Tento test je převzat z literatury [6] a je založen na tom, že při „vhodném“ regulárním výrazu a řetězci, nad kterým se bude vyhledávát, je doba pro vyhledání pro většinu implementací NFA mnohem větší než u DFA [4]. Při tomto testu se vyhodnocuje regulární výraz `X(.+)+X` nad řetezcem `aXcXaaaaaaaaaaaaaaaaaaaaaaaaaaaaa`. Pokud takový test spustíme pomocí dvou různých nástrojů, pro ukázku jsem vybral `GNU awk` a `mawk`, uvidíme veliký rozdíl v rychlosti vyhodnocení. GNU awk používá DFA a vyhodnocení mu trvá na běžném počítači `0,014s`. Mawk naproti tomu používá NFA a jeho čas pro vyhodnocení je `28,004s`.

Příkazy pro ukázku jsem zadal v tomto tvaru:

```
time echo "aXXaaaaaaaaaaaaaaaaaaaaaaaaaaaa" | gawk '/X(.+)+X/ {print}'
time echo "aXXaaaaaaaaaaaaaaaaaaaaaaaaaaaa" | mawk '/X(.+)+X/ {print}'
```

Pokuď bychom chtěli dále rozpoznat, zda se jedná o Tradiční NFA nebo POSIX NFA implementaci, můžeme použít způsob, který byl popsán v odstavci číslo 3.3.

Kapitola 4

Převod PCRE výrazů do POSIX

Tato kapitola obsahuje popis převodu PCRE výrazů do POSIX skupin, jak je prováděn při testování. Zaměřil jsem se na v praxi nejvíce používáne konstrukce, které mělo smyslo brát v úvahu. Při převodu již předpokládám, že nástroje mají implementovanou podporu pro GNU rozšíření.

Jelikož mají regulární výrazy PCRE větší sílu, některé konstrukce nejsme schopni zapsat v POSIX regulárních výrazech. Takto nemůžeme zapsat například všechny konstrukce, které jsou uvozeny metaznaky (`?...`), neboť se jedná o konstrukce, které právě podporuje jen Perl a nástroje od něj odvozené. Těmito metaznaky jsou v Perl uvozeny pojmenované backreference včetně odkazů na tyto backreference, pozitivní i negativní nahlížení a skupiny, které nejsou vyhodnocovány (*Non-Capturing Group*).

Filtrování speciálních konstukcí výrazů Perl a převod tříd znaků je prováděn pro obě skupiny POSIX výrazů shodným způsobem.

4.1 Třídy znaků

Třídy znaků, jak je známe z Perl výrazů (`\w`, `\W`, `\s`, `\S`, `\d`, `\D`) můžeme přepsat do POSIX notace pomocí jim odpovídajících a nám již známých tříd znaků a jejich negací negace. Tento převod je třeba udělat v případě obou skupin POSIX regulárních výrazů.

Třída znaků Perl	Třída znaků POSIX
`\w`	`[[:alnum:]_]`
`\W`	`[^[:alnum:]_]`
`\d`	`[0-9]`
`\D`	`[^0-9]`
`\s`	`[[:space:]]`
`\S`	`[^[:space:]]`

Tabulka 4.1: Prováděné nahrazení skupin jazyka Perl do skupin POSIX.

Při tomto převodu jsem použil nahrazení jak můžeme vidět v tabulce č. 4.1. Nahrazení není ale zcela triviální, neboť je třeba sledovat, zda se daný metaznak vyskytuje ve výčtové množině (např. `[\w]`), nebo se vyskytuje jinde ve výrazu (např. `\w*`). V prvním případě je třeba takový metaznak nahradit řetězcem „`[:alnum:]_`“. V druhém případě musíme ale přidat do nahrazení i uvozující hranaté závorky pro výčtovou množinu – „`[[:alnum:]_]`“.

Tatko probíha nahrazení u všech tříd znaků, které potřebujeme takto nahradit. Obdobně se tak děje i pro negované (velká písmena) třídy znaků.

Na uvedeném příkladu můžeme sledovat i tu skutečnost, že použitím třídy znaků v POSIX přepisu neztratíme podporu pro lokalizaci. Tu bychom ztratili, kdybychom použili přepis za řetězec „`a-zA-Z0-9_`“. Takový přepis by byl správný pouze v rámci množiny ASCII znaků.

4.2 Převod PCRE do BRE

Při tomto převodu je třeba dávat dobrý pozor na to, které znaky mají být escapovány a které ne. Nejdříve escapujeme metaznaky, které jsou přidány podporou GNU rozšířením do BRE – `|`, `?` a `+`. Musíme ale sledovat, zda tyto metaznaky jsou na místě, kde je escapovat máme. Uvažujeme dva případy, kdy takový znak escapovat nebudeme. Pokuď před takovým znakem je již napsáno zpětné lomítko \, znamená to, že právě chceme opačný výraz než je v ERE. Nechceme takový znak použít jako metaznak a jako normální symbol. Takže naopak musíme tento znak \ odstranit před tímto symbolem. A druhý případ je, když je tento znak obsažen ve výčtové množině. Potom takový znak nebudeme escapovat.

Podobně se tomu tak děje s kulatými a složenými závorkami. Platí u nich také to, že pokuď byly uvozeny znakem \, je třeba toto escapování odstranit a zajistit tak požadovaný význam v BRE. A stejně to platí, pokuď jsou závorky obsaženy ve výčtové množině.

4.3 Převod PCRE do ERE

V převodu do ERE není třeba řešit escapování metaznaků ani závorek, jak je to řešeno při převodu do BRE. Tyto metaznaky mají již odpovídající význam, který je shodný jak v notaci PCRE.

Musí se zde ale řešit správné escapování složených závorek, neboť PCRE rozeznává jejich význam podle toho, co je obsaženho uvnitř mezi nimi. Pro nástroje, které používají BRE POSIX výrazy, musíme tento výraz rozpoznat a správně přepsat. Pokuď je v nich rozeznána sekvence znaků, které přestavují reprezentaci intervalu, jsou tyto závorky chápany tak, že uvozují interval. Můžou ale např. stát i takto ve výrazu: `x{abc}x`. Potom by takový regulární výraz odpovídal řetězci: `x{abc}x`. V tomto případě je nutné tyto složené závorky pro BRE escapovat. Složené závorky, které ale uvozují interval, escapovat samozřejmě nesmíme.

Kapitola 5

Implementace

Tako kapitola obsahuje popis implementace, jak byla proveda pro potřebu testování nástrojů, a rozhraní jednotlivých nástrojů. Celá aplikace je implementována v jazyce Python (implementováno a testováno na verzi: 2.4.3) jako sada několika skriptů, nad kterými bylo následně nadstaveno grafické rozhraní.

5.1 Možné nástroje pro měření

Pro měření času běhu jednotlivých nástrojů se nám naskytly dvě možné varianty, které jsme mohli použít pro měření – `perf` a `time`. Oba nástroje mají své výhody a nevýhody, které jsou popsány v následujících podkapitolách

Jelikož bylo stěžejní měření prováděno na školním serveru `ant-3.fit.vutbr.cz`, kde je zastaralejší verze jádra (2.6.18), která nepodporuje nástroj perf, byl zvolen pro měření nástroj time. Díky použití tohoto serveru byl částěčné eliminován problém s aktuálním zatížením systému, který by ovlivňoval výsledky testů. Tento nástroj byl také zvolen z hlediska zvýšení přenositelnosti aplikace.

Ve skriptech je zahrnuta i podpora pro měření nástrojem perf, neboť první testy byly prováděny na stroji s novějším jádrem. Pokud by se tedy použil systém s podporou nástroje perf, dá se přepnout na tento nástroj znegováním jedné proměnné.

5.1.1 Nástroj `perf`

Výkonostní čítače [21] jsou speciální hardwarové registry, které jsou součástí většiny moderních procesorů. Tyto registry umožňují počítat určité hardwarové události, jako jsou například vykonání dílčí instukce, ztráta ve stránkování paměti a podobně. Díky tomu, že se jedná o hardwarové řešení, tak nám tyto statistiky poskytne bez zpomalení jádra systému nebo bez zpomalení samotné aplikace, kterou chceme měřit. I z tohoto důvodu je vhodné využít tento nástroj pro profilování, optimalizování, validaci a měření rychlosti aplikací.

V jádře Linux je zaveden celý podsystém těchto hardwarových registrů, ke kterým lze přistupovat pomocí balíčku nástrojů `perf`. Přes tento balíček jsme schopni pracovat s těmito registry. Nabízím nám statistiky jako např. počet ztrát ve stránkování paměti, přepínání konktextu, přepínání jader procesoru, doba běhu aplikace, počet vykonaných instrukcí, počet cyklů a podobně.

Pro měření jednotlivých nástrojů lze použít nástroj `perf-stat`, který je součástí aplikace `perf`. Tento nástroj jako výstup tiskne na standardní chybový výstup celkovou statistiku běhu aplikace. Příklad takového výstupu můžeme vidět na obrázku č. 5.1. Tento nástroj

nám umožňuje i vícenásobné měření, které můžeme vyvolat přepínačem `-r`, za kterým následuje počet měření. Ve výstupu se nám po té objeví průměrná hodnota všech měření a rozptyl naměřených hodnot udaný v procentech.

```
Performance counter stats for 'ls -al' (5 runs):

     5.055459  task-clock-msecs         #      0.239 CPUs    ( +-   5.689% )
           4  context-switches         #      0.001 M/sec   ( +-  58.998% )
           0  CPU-migrations           #      0.000 M/sec   ( +-    -nan% )
         366  page-faults              #      0.072 M/sec   ( +-   0.055% )
     8350831  cycles                   #   1651.844 M/sec   ( +-   5.431% )
     5741580  instructions             #      0.688 IPC     ( +-   2.162% )
      170562  cache-references         #     33.738 M/sec   ( +-   5.817% )
        5790  cache-misses             #      1.145 M/sec   ( +-  16.745% )

  0.021111776  seconds time elapsed   ( +-  18.682% )
```

Obrázek 5.1: Ukázka výstupu nástroje `perf-stat`.

Celý příkaz pro měření může vypadat následovně: `perf stat -r 5 <COMMAND>`. Často ale chceme do příkazu zapsat složitější konstrukci jako např. přesměrování standradnícho výstupu do souboru nebo použít zřetězení příkazů pomocí roury. Taková konstrukce se provádí zapsaním celého příkazu do pomocného souboru, který je třeba nastavit jako spustitelný. Místo příkazu se potom zadá cesta k tomuto pomocnému souboru.

Nástroj perf je poměrně nová záležitost, která se objevuje v novějších verzích jádra Linux (2.6.32 a výše). Například v distribuci Ubuntu je tento nástroj podporován až od verze 10.04 [15].

5.1.2 Nástroj `time`

Tento nástroj je založen na volání funkce `wait3` popřípadě `waitpid`, pokuď první zmiňovaná není v systému dostupná. Nástroj time spustí aplikaci, která je mu předána jako poslední parametr a po jejím skončení vypíše na standardní chybový výstup statistiku. Pro nás je nejdůležitější reálný čas běhu programu. Přesnější z hlediska nezávislosti testu na aktuální zatížení systému by pro nás bylo sečíst zbývající dvě položky, které jsou údaje o tom, kolik procesorového času strávila aplikace v Kernel módu (`system`) a kolik v uživatelském módu (`user`).

Pokuď ale byla aplikace ukončena záslaním signálu `kill` po doběhnutí timeoutu, v těchto dvou číslech jsou vráceny nuly. Jednodušší řešení tedy je použít položku reálné doby běhu. V případě explicitního ukončení příkazu je tato hodnota 1800 sekund. Díky tomu jsme schopni lehce rozeznat tuto situaci.

5.2 Testování jednotlivých nástrojů

V této podkapitole je popsán způsob, jakým jsou testovány jednotlivé nástroje. Každý nástroj je určený pro rozdílné použití. Z toho důvodu nástroje podporují různé funkce a nastavení. Například nástroj grep pracuje pouze s řádky, takže jej nemůžeme porovnávat

s nástrojem, kterým měříme i víceřádkové výrazy. Nástroje budeme porovnávat vždy jen na odpovídající množině regulárních výrazů.

Při testování jsme se ale vždy snažili o co nejpodobnější význam celého příkazu. U nástroje sed jsme například použili substituci, která odstranila řetězec, který odpovídá danému regulárnímu výrazu. U nástroje grep jsme použili invertované vypisování, takže jsme docílili, že nevypisuje řádky, v kterých byla nalezena shoda s regulárním výrazem. Tyto dva nástroje tedy nezpůsobí naprosto stejné chování, ale chování je co nejvíce podobné, jak nám to tyto nástroje umožnily.

V tabulce číslo 5.1 vidíme verze testovaných nástrojů v této práci.

Nástroj	Verze
PCRE	8.02
Perl	5.12.3
GNU sed/sed -r	4.1.5
GNU grep/egrep	2.5.1
GNU awk	3.1.5
mawk	1.3.4

Tabulka 5.1: Výčet testovaných verzí nástrojů.

5.2.1 PCRE

Pro testování knihovny PCRE [10] se nastkytovalo hned několik možností. Mohli jsme použít nástroje, které jsou přímo součástí knihovny PCRE (`pcretest` nebo `pcregrep`) nebo nějaký program, který používá tuto knihovnu – např. jazyk PHP [25]. Nechtěli jsme ale přidávat další mezivrstvu, která by nám odstiníla co nejpřesnější měření. Nástroj pcretest funguje správně se všemi modifikátory, které podporuje knihovna PCRE, funguje ale pouze řádkově. To znamená, že nemůžeme správně vyhodnotit modifikátor `m`. Proto jsme nakonec pro testování vybrali nástroj `pcregrep`, který je schopne správně obsloužit modifikátory a umožní nám načítat i celý soubor. Navíc má stejný účinek jako nástroj grep, což je vhodné z hlediska zachování co nejvíce podobného chování nástrojů.

Nástroj `pcregrep` ale nepřijímá PCRE výrazy, jak je známe. Žádá si obsluhu modifikátoru v podobě parametrů samotného příkazu. Modifikátor `i` přímo odpovídá parametru `-i`. Obdobně modifikátor `m` odpovídá parametru `-M`, který zařidí jak přepnutí vyhodnocování do *multi-line* módu, tak načítání souboru jako celku. Modifikátor `g` opíšeme parametrem `-o`, který znamená (stejně jak u nástroje `grep`), že je na stadardní výstup vypisována pouze shoda s řetězcem a ne celý řádek. To ale způsobí i globální vyhledávání, protože nástroj s tímto přepínáčem vypíše všechny výskyty v řetězci.

Modifikátory `s` a `U` opíšeme pomocí speciální konstrukce, kterou podporuje knihovna PCRE, kde lze modifikátor zadat přímo do výrazu. Takový modifikátor má potom účinek od místa, kde je uveden, dále. Tyto modifikátory tedy vložíme na začátek výrazu v podobě `(?s)` a `(?U)`.

Ukázka konkrétního příkazu může vypadat takto:

```
../pcre-8.02/pcregrep -vMio -e '(?s)regexpPCRE' data >/dev/null 2>stderr
```

Parametr `-v` způsobuje invertní výpis, takže se vypisují řádky, které neodpovídají danému výrazu. Tím se snažíme napodobit stejné chování jako u předešlého nástroje. Parametr `-e` uvozuje samotný regulární výraz.

Opět můžeme vidět, že pro testování byla použita aktuální přeložená verze knihovny PCRE, která je volána pomocí zadání absolutní cesty. Součástí této knihovny je i zmiňovaný nástroj `pcregrep`.

5.2.2 Perl

Pro jazyk Perl [28] není třeba jakkoliv upravovat regulární výrazy PCRE, protože jsou navzájem kompatibilní s regulárními výrazy jazyka Perl. Musíme jen ošetřit modifikátory, které jazyk Perl nepodporuje. Knihovna PCRE definuje několik modifikátorů, které podporuje jen ona sama, a v jazyce Perl pro ně nemáme obdobu. Tuto situaci řeším jednoduchým filtrováním modifikátorů, kde povolím pouze ty, které podporuje i jazyk Perl (`s`, `m`, `i` a `g`). Modifikátor `U` není v jazyce Perl podporován, tudíž ho nemůžeme připustit. Pokuď se modifikátor `U` vyskytuje v testovaném výrazu, vepíšeme do souboru s naměřenými výsledky poznámku [16].

Zvláštní pozornost je třeba věnovat modifikátoru `m`. Pokuď se vyskytuje v testovaném PCRE výrazu, musíme do příkazu přidat parametr `-00`, který způsobí to, že soubor je načírán celý zároveň a ne po řádcích.

Samotný příkaz může vypadat například takto:

```
../perl-5.12.3/perl -00pe 's/regexpPCRE//smig' <data >/dev/null 2>stderr
```

Je použit příkaz substituce, který se chová stejně jako příkaz substituce například u nástroje sed.

Pro samotné měření jsem použil aktuální verzi, kterou testuji pomocí zadání absolutní cesty, jak je patrné z příkazu. Tato verze je přeložena ve zvláštním adresáři a je součástí aplikace. Jedná se o verzi 5.12.3.

5.2.3 Sed

Nástroj sed [20] je proudový editor pro filtrování a transformace textu. V základním nastavením pracuje s jednotlivými řádky vstupního souboru. Za pomoci registru lze zajistit, aby se soubor načetl a zpracovával celý jako celek. Nejsme ale schopni zajistit úplnou a správnou funkci *multi-line* módu. Znaky pro začátek řádku a pro konec řádku stále odpovídají začátku a konci řetězce. Proto je třeba takové výrazy, které obsahují modifikátor `m` a takový metaznak, vypustit z měření.

Pokuď již načítáme soubor celý zaráz, nástroj sed pracuje vždy v *single-line* módu. Tuto skutečnost nemůžeme jakkoliv ovlivnit, neboť vychází z historie vývoje. Předpokládalo se, že sed bude vždy pracovat pouze s jedním řádkem vstupu a tudíž výraz `.*`, bude vždy znamenat zbytek řetězce do konce řádku. Další data ze vstupu v ten moment vlastně nejsou načtena.

Tento nástroj také nepodporuje nehladovost vyhledávání v jakémkoliv způsobu zápisu. Pro modifikátor `U` nemáme možnost, jak jej přepsat. A stejně tak pro samotné nehladové kvantifikátory (`*?`, `+?` a `??`) neexistuje alternativa zápisu v nástroji sed. Výrazy obsahující alespoň jednu z těchto variant necháme projít testem a do výsledků je pak zapsána poznámka. Nepotřebné otazníky jsou samozřejmě z výrazu odstraněny.

Modifikátory `i` a `g` můžeme ale snadno a přesně přepsat. Tyto modifikátory přidáme za příkaz substituce, kde mají stejný význam jako ve výrazech PCRE.

Úkazka příkazu může vypadat takto:

```
sed -e ':a;N;$!ba;s/regexpBRE//ig' <data >/dev/null 2>stderr
```

Parametr `-e` uvozuje samotný sed příkaz, kde v tomto případě je na začátku zajištěno načítání celého souboru zaráz.

5.2.4 Sed -r

Nástroj sed nám umožňuje přidat parametr `-r` do příkazu, který nám umožní zadávat regulární výrazy v ERE namísto v BRE. Chceme jsme se přesvědčit, zda použití jiné skupiny regulárních výrazů neovlivní rychlost vyhodnocování.

Všechny podporované modifikátory jsou ošetřeny stejným způsobem jak u nástroje sed s BRE výrazy [20].

Příkaz může tedy vypadat například takto:

```
sed -r -e ':a;N;$!ba;s/regexpERE//ig' <data >/dev/null 2>stderr
```

5.2.5 Grep

Primární účel nástroje grep [19] je výběr řádků ze vstupního souboru, které odpovídají zadanému výrazu. Popřípadě lze výpis invertovat pomocí parametru `-v`. Invertovaným výpisem lépe simulujeme podobné chování jako u nástroje sed, kde řetězec, který odpovídá výrazu, odstraňujeme z dat, které vypisujeme na výstup. Z podstaty nástroje samotného nám tedy jasně vyplývá, že nemůžeme pracovat s výrazy přes více řádku, protože nám nedovolí aktuálně načíst a zpracovávat více než jeden řádek. Pokud tedy PCRE výraz obsahuje modifikátor `m`, nejsme schopni takový ekvivalentně použít v nástroji grep.

Modifikátor `U` a nehladové kvantifikátory (`*?`, `+?` a `??`) obsluhujeme stejně jako u nástroje sed. Modifikátor `s` nemá smysl příliš věnovat pozornost, protože pracujeme vždy jen s jedním řádkem, tudíž výraz `.*` vždy odpovídá zbytku do konce řádku. Vyvolání *case-insensitive* vyhledání se provádí pomocí parametru `-i`. Globální vyhledávání `g` je řešeno stejným způsobem jako u nástroje `pcregrep` – parametrem `-o`.

Názorná ukázka příkazu pak vypadá takto:

```
grep -a -o -i 'regexpBRE' data >/dev/null 2>stderr
```

Parametr `-a` zajistí, že grep vyhledává v binárních datech jako kdyby to byla data textová. V základním natavením pro binární data vypíše jen na standardní výstup zda byla nebo nebyla nalezena shoda s těmito daty.

5.2.6 Egrep

Nástroj egrep je pouze alias pro příkaz `grep -E`. Parametr `-E` způsobí, že zadaný regulární výraz je interpretován ve formě ERE místo BRE. Ostatní modifikátory jsou obslouhuvány stejně jako u nástroje grep [19].

Příklad příkazu může tedy vypadat takto:

```
egrep -a -o -i 'regexpERE' data >/dev/null 2>stderr
```

5.2.7 GNU awk

Jedná se o implementaci AWK programovacího jazyka, která byla provedena v rámci projektu GNU. Původní implementace AWK se v dnešní době již často nepoužívá. Nejčastěji se setkáme právě s touto GNU implementací [14]. Nástoj GNU awk se spouští příkazem `gawk`.

U nástroje gawk nastává podobný problém s modifikátorem `m` jaku u nástroje sed. Jsme schopni pomocí nastavení oddělovače řádku a pole načítat soubor celý zaráz místo po řádcích, nejsme ale schopni zajistit správný význam kotev začátku a konce řádku. Proto je třeba takové výrazy, které obsahují tyto kotvy a jsou v módu *multi-line*, vynechat z měření.

Modifikátor `U` a nehladové kvantifikátory zpracováváme stejně jako u nástroju sed. Takový výraz změříme a do výsledku vepíšeme poznámku. Modifikátorem `s` se nebudeme příliš zabývat, protože gawk pracuje vždy v `single-line` módu.

Modifikátor `i` můžeme přepsat za pomoci nastavení hodnoty `IGNORECASE` na nenulovou hodnotu. Tato hodnota se nastavuje v příkazu `BEGIN`, který předchází dalším příkazům. Modifikátor `g` řešíme použitím globálního substitučnícho příkazu `gsub` místo normální substituce `sub`.

Ukázka příkazu:

```
gawk 'BEGIN {FS="\n"; RS=""; IGNORECASE=1} \
{gsub(/regexpERE/, ""); print}' data >/dev/null 2>stderr
```

5.2.8 Mawk

Mawk [2] je stejně jako GNU awk implementace programovacího jazyka AWK. Hlavním rozdílem je to, že je založena na algoritmu POSIX NFA, na rozdíl od gawk, které je založeno na DFA. Tato implementace byla vyvinuta hlavně pro jendoduchost a rychlost [27].

Modifikátory jsou obsluhovány stejným způsobem, jak u gawk, protože se jedná o dvě různé implementace iterpretu stejného jazyka. Nelze ale zpracovávát modifikátor `i`, protože mawk nepodporuje `case-insensitive` mód. Proměnná `IGNORECASE` je totiž GNU rozšířením. Z toho důvodu je mawk nástroj, v kterém můžeme otestovat nejmenší počer výrazů, které máme k testování.

Příklad příkazu může vypadat takto:

```
mawk 'BEGIN {FS="\n"; RS=""} \
{gsub(/regexpERE/, ""); print}' data >/dev/null 2>stderr
```

5.3 Proces testování

Testování každého nástroje probíha ve smyčce, která se provede právě tolikrát, kolik je pravidel v souboru s pravidly. V tomto souboru jsou pravidla uložena každé na zvláštním řádku a jsou v něm uložena ve formě PCRE výrazů. Na začátku smyčky vstupuje aktuálně měřené pravidlo, kde se zkontroluje, zda neobsahuje konstrukce, které aktuální nástroj nepodporuje. Pokud pravidlo nevyhoví, měření se neprovádí a do výsledků je toto pravidlo zapsáno se zápornou hodnotou času, abychom mohli rozlišit, že pravidlo nebylo testováno. Taková pravidla potom nejsou dále zpracovávány do statistik. Pravidla, která daný nástroj, po té převedeme do požadované notace a zajistíme správnou obsluhu modifikátoru PCRE výrazů. Pokuď nástroj používá jiné regulární výrazy než Perl, použijeme pro převod pomocnou funkci `pcre2bre` popřípade `pcre2ere`. Tyto dvě funkce převádí regulární výrazy tak, jak je popsáno v kapitole č. 4.3.

Pro samotné spuštění příkazu je vytvořen pomocný soubor, který obsahuje příkaz včetně další režie. Tento soubor, stejně jako ostatní pomocné soubory, je vytvářen ve složce, která je mapována jako ramdisk a tudíž se nezapisuje na disk. Díky tomu se zrychlí samotný proces testování a není zbytečně využíván pevný disk. Zapouzdření příkazu do souboru nám zajistí, že všechny parametry a přesměrování proudů jsou přiřazeny testovanému příkazu

a ne nástroji pro měření času. Zápis v souboru nám také umožní nastavit maximální čas, po kterém se má příkaz ukončit – time-out. Některé testované příkazy mohou běžet příliš dlouho nebo se dokonce zacyklit. Z toho důvodu je nastavován při testování příkazu daný time-out, po jehož uplynutí je případně takový příkaz ukončen zasláním signálu *kill*.

Celý soubor s příkazem po té může vypadat například takto:

```
timeout()
{
  sleep 1800
  kill -9 \$PID >/dev/null 2>/dev/null
}
../perl-5.12.3/perl -pe 's/a(bc|b)c//i' <data >/dev/null 2>/dev/shm/stderr &
PID=$!
timeout &
wait $PID
```

Samotný příkaz je spuštěn na pozadí a jeho číslo procesu je uloženo do pomocné proměnné `PID`. Zavoláním pomocné námi definované funkce time-out se zajistí čekání po dobu třiceti minut a následné zaslání signálu *kill* procesu s příslušným číslem procesu. Tato funkce ale musí být také volána na pozadí, abychom mohli přejít na řádek s příkazem `wait`. Na tomto řádku se čeká, dokuď není proces s daným číslem ukončen. Ukončen může být buď standardním způsobem, nebo vypršením time-outu. Pokuď příkaz nedoběhl běžným způsobem do konce, ale byl ukončen pomocí příkazu `kill`, jsme tuto skutečnost podle doby běhu programu, která se rovná přesně 1800 sekundám.

Chybový výstup je přesměrováván do pomocného souboru, který je uložen také ve složce mapované na ramdisk. Tento soubor je následně kontrolován, zda neobsahuje chybové hlášení. Některé výrazy v určitých nástrojích běželi určitou dobu a po té skončili chybou. Může se tak například stát u knihovny PCRE při překročení úrovně rekurze. Díky kontrole chybového výstupu jsme schopni tuto skutečnost zaznamenat a takový výraz zapsat do výsledku se záporným časem. Takovy výraz nebude brán na zřetel při následném zpracování dat do histogramu.

5.4 Regulární výrazy pro testování

Regulární výrazy PCRE, které máme k dispozici pro testování, jsme získali z archívu pravidel systému Snort [23]. Aplikace Snort využívá k vyhodnocování PCRE knihovnu, tudíž získané regulární výrazy jsou v PCRE notaci. Z archivu jsme získali 3900 pravidel, které nám slouží jako výchozí množina pro testování. Pro získání pravidel z archivu je vytvořen skript v jazyce Python `dolovanipravidel.py`, kterému se předá jako jediný parametr cesta ke staženému archivu. Skript vygeneruje výsledný textový soubor s pravidly do aktuální složky.

5.5 Testovací data

Data, nad kterými bylo prováděno testování regulárních výrazů, jsem získal za použití nástroje tcpflow. Tento nástroj naslouchá na zadaném síťovém rozhraní a jednotlivé pakety třídí a ukládá podle k sobě souvisejících toků. Tyto toky ukládá do textových souborů, které

obsahují obsah paketů včetně hlaviček [5]. Bylo vhodné mít seskupené k sobě patřící pakety, které dohromady tvoří tok. Testovací data potom byla vytvořena spojením těchto textových souborů do jednoho textového souboru, který měl výslednou velikost zhruba $318MB$. Při zachytávání paketů při vytváření testovacích dat jsem se snažil simulovat běžný způsob procházení internetu. Bylo načteno několik stránek s různým počtem a velikostí obrázku a bylo přehráno několik videí.

5.6 Vytváření histogramu

Pro vytvoření histogramu je používán jednoduchý skript napsaný v jazyce Python. V tomto skriptu se prochází soubor s naměřenými daty a vytváří se potřebná tabulka s četností časů jednotlivých výrazů v intervalech. Skript umožňuje snadné omezení maximální a minimální hodnoty na ose x a počet intervalů, na které má být histogram spočítán. Na závěr zpracování výsledků je sestaven potřebný profil pro nástroj gnuplot a za pomoci gnuplotu je vygenerován výsledný histogram. Pro zajištění kompatibility je použit opět přeložený gnuplot, který je spuštěn pomocí zadání absolutní cesty. Je použita aktuální verze 4.4.3.

Pro zlepšení znázornění naměřených dat omezuji maximální hodnotu osy x většinou na 120 sekund při vytváření histogramů do této práce. Jedna výrazně vysoká hodnota způsobí, že zbytek dat může padnout do jednoho intervalu a takový histogram po té nemá žádnou výpovědní váhu. Vytváření histogramů je proto vždy vhodné zkontrolovat a ručně opravit parametry.

Skript `histogram.py` je spuštěn buď s jedním paramatrem, který definuje cestu k textovému souboru s naměřenými výsledky, nebo se třemi, kde první je opět soubor s výsledky a další dva definuji minimum a maximum intevalu na ose x.

Kapitola 6

Teoretické limity

V této kapitole jsou popsány případy, pro které by různé varianty implementace měli být nejpomalejší. Pro nástroje založené na DFA se řeší problém regulárních výrazů, které způsobují zpomalení při determinizaci. Pro nástroje založené na implementaci NFA budou popsány výrazy, které mohou uchovávat velké množství aktivních stavů.

Oba přístupy mají svoje výhody a nevýhody. Proto se v dnešní době nejčastěji setkáme s přístupem, který obě tyto varianty kombinuje. Takové systémy jsou označovány jako hybridní. Existuje více možností, jak oba přistupy zkombinovat. Vždy je snaha eliminovat nedostatky daného nástroje a dosáhnout tak co nejlepších výsledků. Jeden přístupů je popsán v článku [1].

6.1 Nejhorší případ pro DFA

Pro implementaci založenou na DFA můžeme najít několik případů, které zpomalují nebo omezují funkci této implementace. Tyto případy vychází z toho, že je třeba před samotným vyhodnocováním provést determinizaci konečného automatu. Při této determinizaci může být teoreticky nárůst DFA stavů až exponenciální. To může vést k nepoužitelnosti takového nástroje pro řešení běžných problémů.

Velký nárust stavů zpravidla způsobuje několik určitých konstrukcí. První varianta je alternace dvou regulárních výrazů RV_1 a RV_2. Pro tyto dva regulární výrazy bychom sestrojili dva automaty DFA_1 a DFA_2, které by jim odpovídali. Pokuď tyto dva výrazy dáme do alternace a budeme chtít vytvořit automat DFA_{12}, jeho počet stavů bude větší než soušet stavů DFA_1 a DFA_2. Popřípadě automat DFA_{12} nepůjde sestrojit vůbec, díky exponenciálnímu nárustu stavů.

Další problém může způsobit výraz „tečka-hvězdička“. Pod tímto pojmem si představme i výrazy $[\hat{}c_1c_2 \ldots c_k]*$. Tento výraz reprezentuje řětěc, který obsahuje libovolný počet libovolných symbolů kromě $c_1c_2 \ldots c_k$. Toto omezení symbolů přidává ještě další nutnou režii a zvyšuje počet stavů. Výrazy „tečka-hvězdička“ způsobují veliký nárust stavů, pokuď jsou použity v alternaci s dalším výrazem.

Další konstrukce, které ovlivňuje počet DFA stavů je interval. Při použití intervalu ve výrazu může dojít k exponenciálnímu nárůstu stavů i bez alternace s pouze jedním regulárním výrazem. Nejvíce počet stavů naroste, pokuď je interval aplikován na `tečku`. Například pro výraz `prefix.{100}suffix`, který znamená kladné vyhodnocení v případe, že mezi `prefix` a `suffix` bude přesně 100 symbolů, může být použito přes milion DFA stavů [1].

6.2 Nejhorší případ pro NFA

Časově náročným pro NFA může být každý příkaz, který způsobí, že je třeba uchovávat mnoho aktivních stavů. V nejhorším případě může dojít i k problémům s nedostatkem přidělené paměti. Pokud by nastala situace, že budou všechny stavy n_{NFA} aktivní, znamenalo by vyhodnocení přechodu pro jeden přijatý symbol n_{NFA} přístupů do paměti [1].

Počet aktivních stavů je závislý na vstupním řetězci, který je vyhodnocován. Pro názorný příklad si uveďme konečný automat na obrázku č. 6.1. Pokud takový nedeterministický konečný automat bude přijímat řetězec `bbbbbbbb`, bude aktivní pouze jeden stav a to stav počáteční s. Při vstupním řetězci `aaaaaaaa` budou po přijetí třetího symbolu aktivní stavy čtyři s, q_1, q_2 a q_3.

Obrázek 6.1: Jednoduchý konečný automat pro ukázku závislosti počtu aktivních stavů na vstupním řetězci.

V případě IDS systému se tato skutečnost stává nebezpečnou, protože vstupní data může ovlivnit uživatel a tak by mohl popřípadě způsobit zpomalení vyhodnocování. U DFA je jeho nejhorší případ závislý pouze na regulárním výrazu. Předem jsme tedy schopni určit, jak dlouho bude vyhodnocení trvat, pouze se znalostí délky vstupních dat.

V dnešní době se stává pro NFA největším zdržením přídávání podpory nových funkcí. Podpora těchto funkcí přidává další režii, která mnohdy dokáže zpomalit vyhodnocování. O této problematice pojednává např. článek [4], kde jsou porovnávány nástroje s vlastní referenční implementací NFA.

Příkladem výrazu, který způsobí mnoho aktivních stavů je právě např. výraz `X(.+)+X`, který je uveden a použit v odstavci 3.4.

Kapitola 7

Výsledky měření

Tato kapitola obsahuje výsledky měření jednotlivých nástroju a jejich srovnání s odpovídajícími výsledky jiných nástrojů. Srovnávání výsledků je prováděno ve třech skupinách, protože jednotlivé nástroje se liší ve své síle a v tom, jaké konstrukce podporují. Liší se tedy množina regulárních výrazů, které dokážou vyhodnotit. Musíme proto srovnávat nástroje, které jsou si co možná nejpodobnější a vyhodnocují stejnou množinu pravidel.

Do dalších dvou skupin, kde už je omezená množina výrazů, přidáme ještě výsledky měření pro PCRE, které budou vyhodnocovány nad příslušnou množinou výrazů. Tím získáme alespoň určitou možnost porovnání.

Regulární výrazy, které vyhodnocujeme, jsme získali z archivu pravidel aplikace Snort [24] a testovací data, nad kterými se vyhledává, jsme získali záznamem zhruba hodinového sledování provozu na notebooku v domácnosti při běžném užívání internetu. Více o těchto zdrojích v odstavcích č. 5.4 a 5.5.

7.1 Celá množina pravidel

Celou množinu regulárních výrazů PCRE, kterou máme k dispozici k měření, dokáží změřit pouze ty nástroje, které podporují regulární výrazy jazyka Perl. Z testovaných nástrojů jsou to tedy nástroje pcregrep a Perl.

7.1.1 PCRE

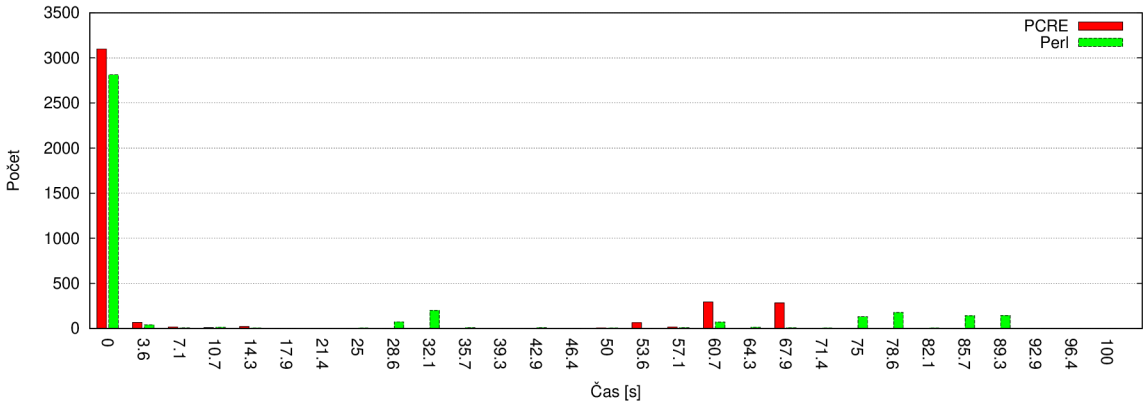
Knihovna PCRE byla jediný nástroj, kde všechna pravidla doběhla do 30 minut. Ani jeden test nebyl ukončen explicitně zasláním signálu *kill*. Na obrázku č. 7.1 vidíme histogram měření v intervalu 0 až 100 sekund. Do tohoto intervalu spadla většina pravidel – 3882, to znamená, že pouze 18 jich běželo déle než 100 sekund. Nejdelé běžící regulární výraz trval 1487.89 sekund a byl to výraz, který začínal podvýrazem `[^\x3b\x3a\r\n]*` následovaným skupinou s několikanásobnou alternací. Interval 0 až 100 sekund byl zvolen úmyslně, abychom měli stejný interval pro srovnání s nástrojem Perl, kde je využití tohoto intevalu lepší.

Většinu výrazů můžeme pozorovat v prvním intervalu cca do 5 vteřin. Přiblížení na tento interval můžeme vidět na obrázku č. 7.2. Do tohoto intervalu padlo celkem 3156 výrazů.

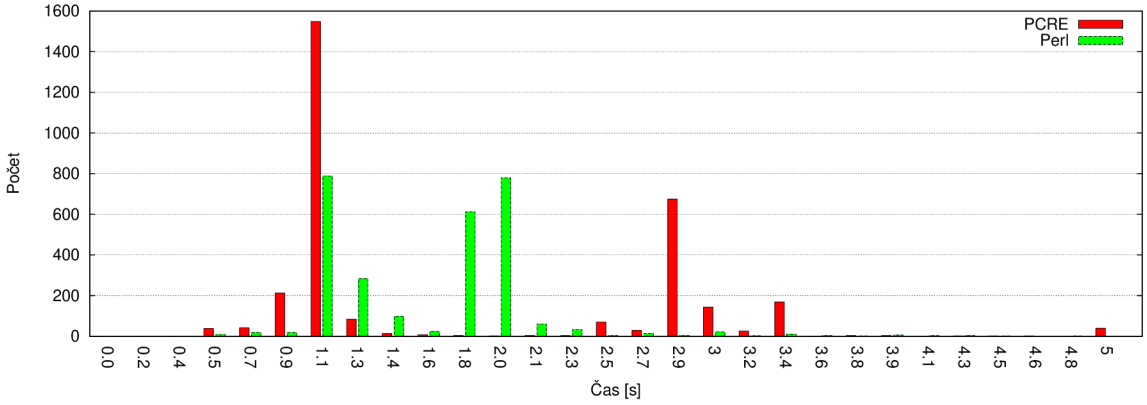
7.1.2 Perl

Při měření nástrojem Perl byly dva výrazy ukončeny po vypršení time-outu. Do intervalu 0 až 100 sekund se vešlo celkem 3887 pravidel, takže pouze 13 výrazů trvalo delší dobu. Tento interval můžeme vidět na obrázku č. 7.1. Pokuď vypustíme dva výrazy, které běželi déle než time-out, třetím nejdéle trvajícím výrazem byl výraz, který začínal podvýrazem `[^\x26\x20\x0a]*`. Jeho naměřený čas byl 662.58 sekund. V tomto výrazu se dále opakuje stejná konstrukce iterované negované množiny ještě dvakrát.

Na obrázku 7.2 můžeme vidět interval od 0 do 5 vteřin pro jazyk Perl. Do tohoto intervalu se v případě jazyka Perl vešlo 2840, cože je o 316 výrazů méně než u knihovny PCRE.



Obrázek 7.1: Výsledky měření PCRE a Perl pro pravidla s časem do 100 sekund.



Obrázek 7.2: Výsledky měření PCRE a Perl pro pravidla s časem do 5 sekund.

7.1.3 Srovnání

Z histogramů na obrázku č. 7.1 zobrazujících výsledky v čase do 100 sekund si můžeme povšimnout, že v případě PCRE máme skupinu zhruba 600 výrazů kolem času 60 až 70 vteřin. Tato skupina stejný výrazů se u jazyka Perl přesouvá k hodnotám 75 až 90 vteřin. Kolem času 60 vteřin nám zůstává cca 70 výrazů. U jazyka Perl dále můžeme pozorovat skupinu zhruba 270 výrazů, které se pohybují kolem hodnoty 30 vteřin. Tato skupina výrazů

se v PCRE vyhodnocuje do 10 vteřin. Existují ale výrazy, které v jazyce Perl běžely rychelji než v knihovně PCRE. Celkově je ale Perl pomalejší.

Pokud se ale podíváme na histogramy na obrázku č. 7.2, které zobrazují interval do 5 sekund, vidíme, že Perl mnohem více výrazů dokáže vyhodnotit do 2 sekund. Nesmíme zapomínat, že na histogramu pro Perl je zobrazeno o 316 výrazů méně. Tyto výrazy trvali déle než 5 sekund. U PCRE se nám objevule výrazná skupina zhruba ticíse výrazů, která je umístěna kolem hodnoty 3 sekund. Tyto výrazy byly rychlejší v jazyce Perl.

7.2 Množina POSIX výrazů

Tato kapitola popisuje výsledky testů pro nástroje, které podporují regulární výrazy POSIX a dokáží pracovat i víceřádkově. Tyto podmínky splňuje nástroj sed. Z původního počtu 3900 pravidel nám pro měření těchto nástrojů zůstalo 1311 výrazů. V těchto výrazech máme pouze ty výrazy, které neobsahují nepodporované konstrukce a nekončí chybou vinou chybného zápisu regulárního výrazu.

7.2.1 Sed

Šest regulárních výrazů neskončilo v čase do 30 minut. Ve všech šesti případech regulární výraz začíná podvýrazem `[^c]{n}`, kde c je symbol nebo více symbolů (zpravidla ne více než čtyři) a n je vyšší číslo (např. 512). Tato konstrukce byla testováná i pro delší čas. Test běžel přes 12 hodin a stále neskončil s úspěšným koncem.

Na obrázku č. 7.3 vidíme histogram výsledků, které měli čas běhu menší než 120 sekund. Na tomto obrázku je celkem 1285 výsledků, což znamená, že 20 jich běželo déle a 6 nedoběhlo dokonce.

Obrázek č. 7.4 zobrazuje výsledký nástroje sed pro interval 0 až 15 sekund. Do tohoto intervalu padlo celkem 1034 měření regulárních výrazů.

7.2.2 Sed s parametrem -r

U nástroje sed s přepínačem -r šest pravidel nedoběhlo v časovém time-outu. Bylo to šest stejných pravidel jako u nástroje sed bez tohoto přepínače. Histogram výsledků můžeme vidět na obrázku č. 7.3. V případě tohoto nástroje se vešlo do tohoto intervalu všech 1305 zbývajících pravidel.

Na obrázku č. 7.4 můžeme vidět histogram výsledků přiblížený na interval 0 až 15 sekund. V histogramu je zobrazeno celkem 1053 výrazů.

7.2.3 GNU awk

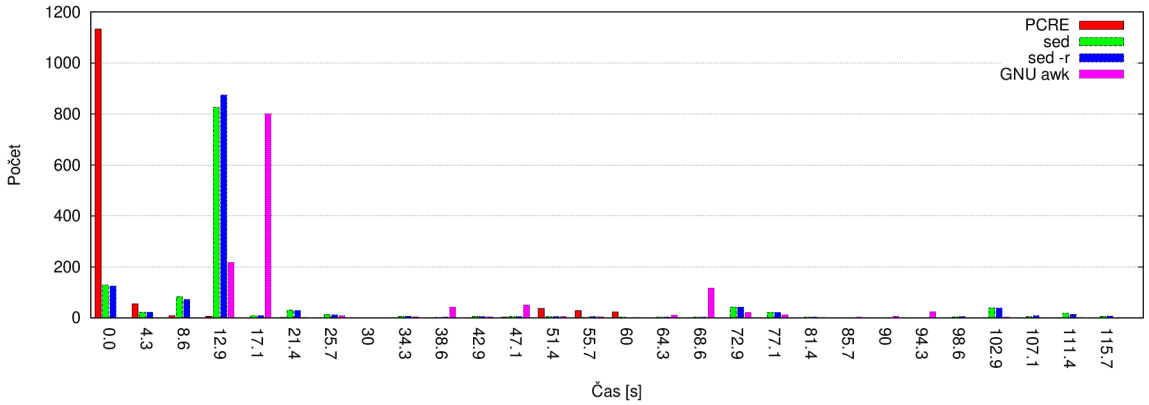
Výsledky pro nástrooj GNU awk (obrázek č. 7.3) jsou zvláštní na první pohled tím, že žádný výraz nebyl vyhodnoce rychleji než za 15 vteřin. Z toho důvodu zde není uveden histogram pro tyto hodnoty. Všechny výrazy doběhli v požadovaném time-out 30 minut.

7.2.4 PCRE

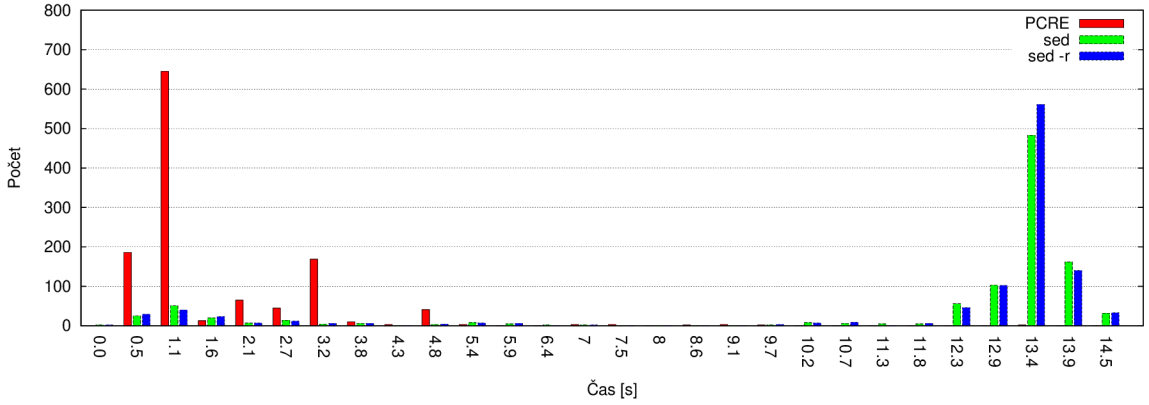
Abychom mohli porovnat tyto nástroje s knihovnou PCRE, na obrázku č. 7.3 vidíme její histogram výsledku pro interval 0 až 120. Tento histogram obsahuje spoustu nulových sloupců, což by se u histogramu nemělo vyskytovat. Interval jsem ale nechal stejný, aby

na první pohled bylo vidět srovnání s knihovnou PCRE. Na tomto histogramu můžeme vidět 1301 hodnot.

Na histogram pro interval 0 až 15, který můžeme vidět na obrázku č. 7.4, je opět zachován stejný interval pro rychlé srovnání s nástrojem sed. Do tohoto intervalu padlo 1203 naměřených hodnot.



Obrázek 7.3: Výsledky měření PCRE, sed, sed -r a gawk pro pravidla s časem do 120 sekund.



Obrázek 7.4: Výsledky měření PCRE, sed a sed -r pro pravidla s časem do 15 sekund.

7.2.5 Srovnání

Z obrázku č. 7.3 vidíme, že nástroj sed má v obou případech velmi podobný průběh. Zhruba 60 výrazů kolem hodnoty 75 sekund a další skupina zhruba 70 výrazů mezi hodnotami 100 až 115 sekund. O hodnotách menších než 15 vteřin se budeme zabývat až z dalších histogramů, protože uvidíme přesnější průběh. Z obrázku výsledků pro PCRE ale vidíme pouze jednu skupinu 90 výrazů v intervalu 50 až 60 sekund. Zbytek výrazů po té skončilo s časem menším než 10 vteřin. GNU awk je v porovnání s ostatními dvěmi nástoji nejpomalejší. Ani nejrychleji vyhodnocený výraz tímto nástrojem nemá čas běhu pod 15 vteřin.

Při zaměření se na výsledky v čase menším než 15 sekund (obrázek č. 7.4) vidíme opět velmi podobný průběh pro obě varianty nástroje sed. Naprostá většina výsledků se pohybuje v intervalu 12 až 15 vteřin. Při porovnání těchto výsledků s výsledky PCRE vidíme hned

na první pohled, že nástroj sed je pomalejší. V případě PCRE spadá naprostá většina do intervalu 0 až 4 sekundy. Naproti tomu v nástroji sed se do tohoto intervalu vejde zhruba 125-krát v obou variantách.

7.3 Množina jednořádkových pravidel

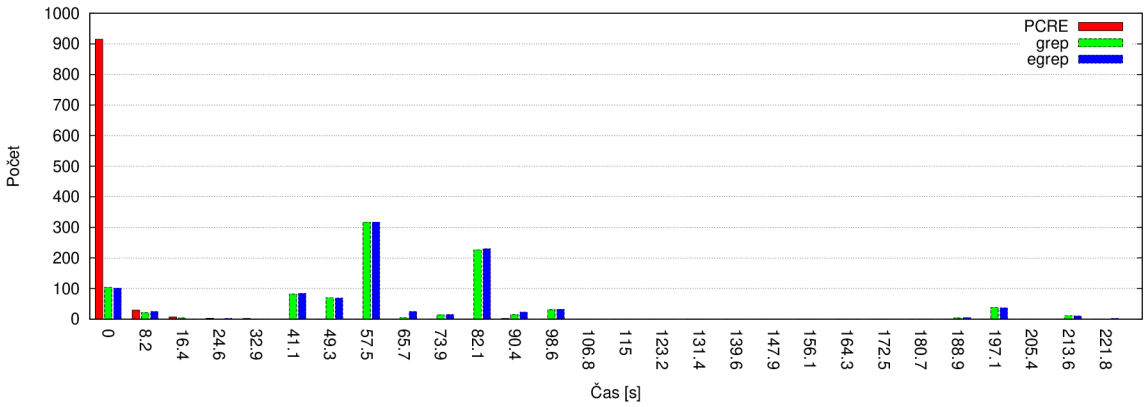
Jelikož nástroj grep nedokáže načítat více než jeden řádek ze vstupního souboru, můžeme ním měřit pouze regulární výrazy, které jsou řádkové. Výchozí množina se tím omezí na 961 výrazů pro testování. Měření těchto výrazů provádíme nástroji grep a egrep a pro srovnání nástrojem pcregrep.

7.3.1 Grep

Pro zobrazení výsledku z měření nástroje grep není třeba vkládat dva histogramy. Při pohledu na obrázek č. 7.5 vidíme, že většina naměřených hodnot se pohybuje velmi vysoko a není třeba se podrobněji dívat na údaje blížící se nule. Důležité je zmínit, že 5 výrazů bylo třeba ukončit po uběhnutí časového limitu půl hodiny. Tyto výrazy mají stejnou stavbu jako výrazy, které shodně nedoběhly u nástroje sed. Na zobrazeném histogramu vidíme celkem 948 hodnot.

7.3.2 Egrep

U nástroje egrep je situace velmi podobná jako u nástroje grep. I zde pět výrazů nedoběhlo v požadovaném časovém limitu do konce. Na obrázku 7.5 můžeme vidět zobrazené výsledky, které jsou téměř stejně rozložené jako u nástroje grep.



Obrázek 7.5: Výsledky měření PCRE, grep a egrep pro pravidla s časem do 230 sekund.

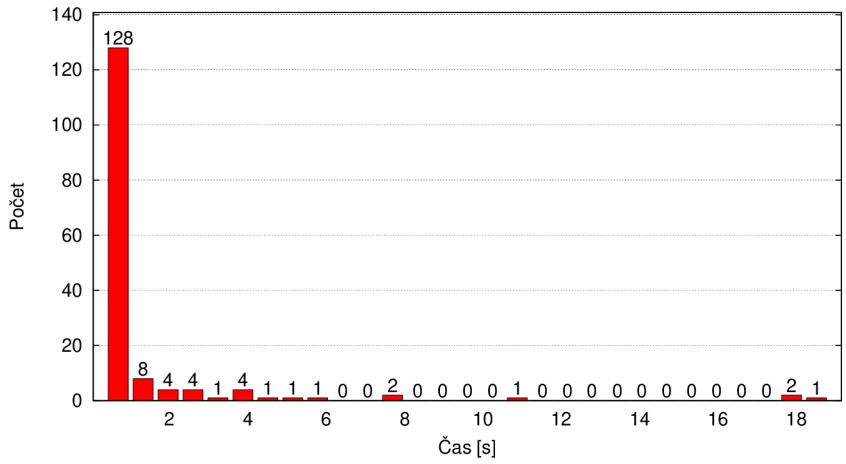
7.3.3 PCRE

Výsledky pro knihovnu PCRE pro danou množinu výrazů jsou zobrazeny v obrázku č. 7.5. Naprostá většina výrazů – 909 – byla vyhodnoce v čase menším než 6 sekund a z toho dokonce 792 výrazů v čase do 2 vteřin. Dalším důležitým poznatkem je to, že všechny výrazy doběhly do konce. Žádné měření tedy nebylo třeba ukončit zasláním signálu.

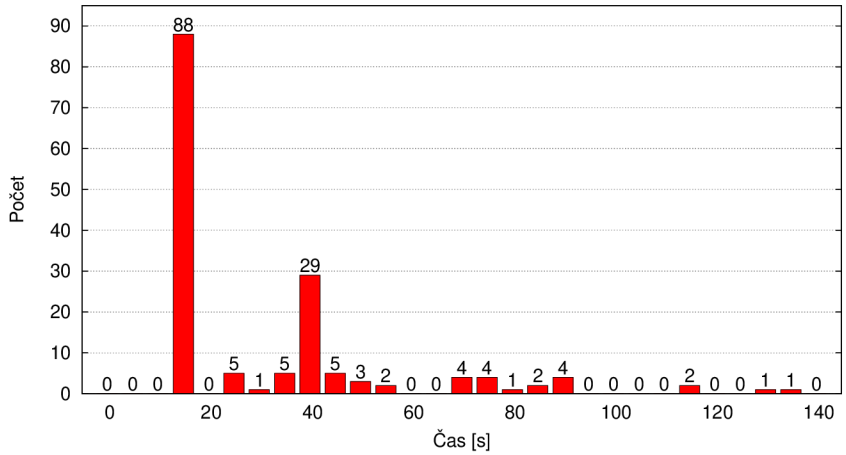
7.3.4 Srovnání

Výsledky (obrázek č. 7.5) nástroju grep a egrep jsou téměř totožné. Z toho vyplývá, že v případě těchto nástrojů neovlivní rychlost to, zda použijeme BRE nebo ERE a k tomu příslušný nástroj. Pokuď ale chceme porovnávat výsledky s knihovnou PCRE, vidíme, že tato knihovna je mnohem rychlejší.

7.4 Mawk



Obrázek 7.6: Výsledky měření nástroje mawk.



Obrázek 7.7: Výsledky měření nástroje GNU awk nad stejnou množinou výrazů jako mawk.

Pro zajímavost je v této podkapitole uvedeno porovnání výsledku naměřených nástrojem mawk s nástrojem GNU awk. Jelikož mawk neumožňuje možnost *case-insensitive* vyhledávání, omezí se nám testovací množina výrazů na počet 158. To již není příliš reprezentativní vzorek dat. Mohou v těchto výrazech chybět některé konstrukce, které by ovlivnili výsledky testování.

Na obrázku č. 7.6 vidíme výsledky naměřené nástrojem mawk. Porovnáváme je s výsledky na obrázku č. 7.7, které jsou naměřeny nástrojem GNU awk nad stejnou množinou výrazů.

Nástoj mawk se jeví v tomto porovnání jako velmi rychlá implementace oproti implementaci nástroje GNU awk. Všechny výrazy mají v nástroji mawk čas běhu do 18 vteřin, zatímco v GNU awk se pod tento čas vyhodnotilo pouze 86 výrazů. Nejpomalejší výraz v GNU awk běžel 201 sekund. Díky tomu nástroj mawk jistě stojí za povšimnutí.

Kapitola 8

Závěr

V této práci jsme provedli srovnání rychlosti několika nástrojů pro vyhledávání regulárních výrazů. Různé nástroje používají pro zápis regulárních výrazů odlišné skupiny. Z toho důvodu byl vytvořen automatický převod z regulárních výrazů PCRE do dalších skupin BRE a ERE. Regulární výrazy, které obsahují nepřevoditelné konstrukce, jsou z měření vyřazeny. Pro snadnější provádění samotného testování nástrojů bylo implementováno grafické rozhraní, které dovoluje definovat soubor s regulárními výrazy v PCRE notaci, vstupní soubor s daty a výběr testovaných nástrojů.

Nejdůležitějším zjištěním bylo ověření velmi dobrých výsledků implementace knihovny PCRE. Dosáhla lepších výsledků nad celou množinou PCRE výrazů, které jsme měli k dispozici, ve srovnání se skriptovacím jazykem Perl. Její implementace je založena na NFA a tudíž je její rychlost závislá na vstupních datech. Knihovna PCRE je rychlejší i v porovnání s nástroji GNU sed, GNU awk s GNU grep na odpovídájí množině regulárních výrazů. Nejhorších výsledků naopak dosáhl nástroj GNU awk, který je velmi pomalý i pro nejjednodušší výrazy. Toto zpomalení je zřejmě způsobeno hybridním přístupem, který tento nástroj používá.

V rámci dalšího rozvoje by bylo možné rozšířit množinu regulárních výrazů pro testování o výrazy, které reprezntují nejhorší případy pro jednotlivé implementace. Také bychom se mohli podrobněji zabývat různými hybridními přístupy a pokusit se pro ně stanovit nejhorší případy.

Další možné rozšíření by bylo zvětšení počtu nástrojů, které testujeme a porovnáváme. Vhodné by bylo přidat nástroj, který je založen na „čistém“ DFA a porovnat s ním výsledky. Implementací jazyka awk je v dnešní době dostupných mnoho (nawk, mks awk, tawk, awka, jawk). Tyto implementace by bylo vhodné porovnat mezi sebou a hlavně s implemtací GNU awk, která je nejrozšířenější a velmi pomalá. Další nástroje, které by bylo vhodné přidat pro srovnání, jsou takové, co by podporovali regulární výrazy jakyka Perl jako například Python, Java, Ruby nebo nástroj GNU grep s parametrem `-p`.

Budoucí práce v oblasti implementace se bude zabývat rozšířením grafické aplikace o možnost zobrazování grafů přehledně přímo v této aplikaci.

Literatura

[1] Becchi, M.; Crowley, P.: A hybrid finite automaton for practical deep packet inspection. CoNEXT '07, New York, USA: ACM, 2007, ISBN 978-1-59593-770-4, http://doi.acm.org/10.1145/1364654.1364656.

[2] Brennan, M.: Man mawk. http://www.bash-linux.com/unix-man-mawk.html, 1994-12-22 [cit. 2011-05-13].

[3] Chytil, M.: *Automaty a gramatiky.* Nakladatelství technické literatury, 1984, typové číslo L11-E1-IV-41f/11 878.

[4] Cox, R.: Regular Expression Matching Can Be Simple And Fast. http://swtch.com/ rsc/regexp/regexp1.html, 2007-01 [cit. 2011-05-04].

[5] Elson, J.: Manpage of tcpflow. http://www.circlemud.org/~jelson/software/tcpflow/tcpflow.1.html, 2001-03-01 [cit. 2011-05-12].

[6] Friedl, J. E. F.: *Mastering Regular Expressions, 3rd Edition.* O'Reilly, 2006, ISBN 10-0-596-52812-4.

[7] Goyvaerts, J.: GNU Regular Expression Extensions. http://www.regular-expressions.info/gnu.html, 2010-09-22 [cit. 2011-05-04].

[8] Goyvaerts, J.: Regex Tutorial - Atomic Grouping. http://www.regular-expressions.info/atomic.html, 2011-02-23 [cit. 2011-05-11].

[9] Hazel, P.: PCRE – Perl-compatible regular expressions – man pages. http://www.pcre.org/pcre.txt, 2010-01-03 [cit. 2011-05-08].

[10] Hazel, P.: PCRE - Perl Compatible Regular Expressions. http://www.pcre.org/, 2011-01-15 [cit. 2011-04-05].

[11] Hazel, P.: Differences from Perl. http://en.wikipedia.org/wiki/Perl_Compatible_Regular_Expressions #Differences_from_Perl, 2011-03-25 [cit. 2011-04-22].

[12] Herold, H.: *awk & sed, Příručka pro dávkové zpracování textu.* Computer Press, 2004, ISBN 80-251-0309-9.

[13] Hopecroft, J. E.; Motwani, R.; Ullman, J. D.: *Aoutomata Theory, Lanquages and Computation.* Pearson Education, 2007, ISBN 0-321-47617-4.

[14] Kernighan, B.: Man page gawk section 1. http://www.manpagez.com/man/1/gawk/, 2010-05-13 [cit. 2011-05-11].

[15] Kirkland, D.: Ubuntu Manpage: perf – Performance analysis tools for Linux. http://manpages.ubuntu.com/manpages/lucid/en/man1/perf.1.html, 2010 [cit. 2011-05-10].

[16] Kolektiv autorů: Perl Command-Line Options. http://www.perl.com/pub/2004/08/09/commandline.html, 2004-08-10 [cit. 2011-05-13].

[17] Kolektiv autorů: Basic Regular Expressions. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html#tag_09_03, 2004 [cit. 2011-05-04].

[18] Kolektiv autorů: Extended Regular Expressions. http://pubs.opengroup.org/onlinepubs/009695399/basedefs/xbd_chap09.html#tag_09_04, 2004 [cit. 2011-05-04].

[19] Kolektiv autorů: Man page grep section 1. http://www.manpagez.com/man/1/grep/, 2009-02-13 [cit. 2011-05-13].

[20] Kolektiv autorů: Man page sed section 1. http://www.manpagez.com/man/1/sed/, 2009-05-01 [cit. 2011-05-13].

[21] Kolektiv autorů: Perf Wiki. https://perf.wiki.kernel.org/index.php/Main_Page, 2009-09-24 [cit. 2011-03-29].

[22] Kolektiv autorů: POSIX. http://en.wikipedia.org/wiki/POSIX#cite_note-doc-2, 2011-03-24 [cit. 2011-04-01].

[23] Kolektiv autorů: Snort :: snort-rules. http://www.snort.org/snort-rules/, 2011-04-23 [cit. 2011-05-13].

[24] Kolektiv autorů: Snort :: Requirements. http://www.snort.org/start/requirements, 2011-05-10 [cit. 2011-05-11].

[25] Kolektiv autorů: PHP: Introduction Manual. http://www.php.net/manual/en/intro.pcre.php, 2011-05-13 [cit. 2011-05-14].

[26] Kozen, D. C.: *Automata and Computability*. Springer Science + Business Media, 1997, ISBN 0-387-94907-0.

[27] O'Connor, B.: Don't MAWK AWK – the fastest and most elegant big data munging language! http://brenocon.com/blog/2009/09/dont-mawk-awk-the-fastest-and-most-elegant-big-data-munging-language/, 2010-04-30 [cit. 2011-05-11].

[28] Wall, L.: Perl documentation. http://perldoc.perl.org/perl.html, [cit. 2011-05-13].

Dodatek A

Obsah CD

Přiložené CD obsahuje zdrojové soubory aplikace.