

Vysoké učení technické v Brně

Fakulta informačních technologií



Projektová praxe

Minimalizace automatů pro inspekci síťového provozu

25. ledna 2018

Martin Dvořák
vedoucí: Ing. Ondřej Lengál, Ph.D.

Obsah

1	Úvod	3
2	Hledání vhodných stavů konečného automatu	4
2.1	Konečný automat	4
2.2	Bezkolizní relace	4
2.3	Algoritmus 1: Tvorba bezkolizní relace	4
2.3.1	Popis	4
2.3.2	Popis činnosti	5
2.3.3	Příprava	6
3	Shlukování stavů	7
3.1	Algoritmus 2: Hledání úplných podgrafů	7
3.1.1	Popis	7
3.1.2	Popis činnosti	8
3.1.3	Příprava	9
4	VHDL reprezentace konečného automatu	10
4.1	Realizace bez kompaktní reprezentace	10
4.2	Realizace s kompaktní reprezentací	10
5	Experimenty	11
5.1	Experiment nad Algoritmem 1	11
5.2	Experiment nad Algoritmem 2	11
5.3	Experiment nad VHDL reprezentací	12
6	Závěr	13

1 Úvod

U monitorování síťového provozu se konečné automaty používají k detekci protokolů, útoků nebo k identifikaci osob. Tyto automaty se získávají pomocí regulárních výrazů. Předfiltrování pomocí regulárních výrazů je nejpoužívanější, nicméně roste její zdrojová náročnost, a to kvůli růstu rychlosti sítě a komplexnosti regulárních výrazů. Běžný program pro monitorování síťového provozu je schopný zvládat rychlost sítě do 1Gbps. Tyto softwary nemohou monitorovat tok dat v páteřních sítích, jelikož rychlost se zde pohybuje mezi desítkami až stovkami Gbps.

Jako vhodné řešení se jeví použití alespoň částečné implementace konečného automatu na hardware. Hardware potom plní činnost předfiltrování, a tedy velké množství paketů odstraňuje před dalším zpracováním. Tím snižuje objem dat sítě, a běžný software pro monitorování síťového provozu je shopen s těmito předfiltrovanými daty pracovat. Nejpoužívanější technologií pro toho předfiltrování je v současné době FPGA. Díky paralelismu dovolují FPGA čipy efektivně implementovat nedeterministické konečné automaty, které vzniknou ze vstupních regulárních výrazů. I přestože se FPGA čipy stále zvětšují, rychlost páteřních sítí stoupá rychleji. Proto je nutné zavést paralelismus na čipu FPGA. V každé paralelní větvi se simuluje činnost jiného konečného automatu. Vedlejší účinek paralelismu je zvětšení plochy čipu. Z tohoto důvodu je redukování velikosti konečných automatů na čipech FPGA velice důležitým úkolem. [1]

Běh konečného automatu je reprezentován na FPGA čipu tak že, konkrétní stav se aktivuje (tzn. nastavení registru), pokud je vstupní signál roven logické jedničce. Tu lze získat kombinační logikou, ve které se nachází vstupní symboly a stavy konečného automatu. Pokud je zpracováván vstupní symbol a správný stav je právě aktivní (určuje množina přechodů konečného automatu), je provedeno nastavení logické jedničky na vstupní signál, signál je zapsán do registru v dalším taktu obvodu.

Velikost konečných automatů je třeba redukovat na úrovni hardwarové reprezentace počtem použitých komponent. V první řadě se jedná o počty registrových pamětí, kde jednobitový registr reprezentuje určitý stav konečného automatu. Redukci je nutno provést tak, aby nebyla značně snížena maximální frekvence obvodu (tzn. aby obvod mohl být stále použit v prostředí s vysokými rychlostmi síťové komunikace).

Pokud je možné sdílet jeden paměťový registr pro více stavů, pak lze takto redukovat plochu hardware. Pomocí binárního kódování je možné do n -bitového registru zakódovat až $2^n - 1$ stavů. Nutné rezervovat jednu hodnotu (rezervována hodnota 0 binárně) pro možnost, kdy není aktivní žádný stav z tohoto registru. Je nutné, aby byl aktivní maximálně jeden stav ze stavů, které jsou reprezentovány jedním registrem. Tyto stavy musí být mezi sebou bezkolizní.

Problém, který zde nastává, je rozeznání stavů konečného automatu tak, aby nedošlo ke kolizi v n -bitové registrové paměti.

2 Hledání vhodných stavů konečného automatu

2.1 Konečný automat

V této práci je konečný automat pětice $M = (Q, \Sigma, R, S, F)$.

Kde

- Q reprezentuje konečnou množinu všech stavů.
- Σ je vstupní abeceda.
- $R : Q \times \Sigma \rightarrow 2^Q$
- S je množina počátečních stavů, $S \subseteq Q$.
- F je množina koncových stavů, $F \subseteq Q$.

Konfigurace konečného automatu M je řetězec $\chi \in Q\Sigma^*$

Nechť pax a qx jsou dvě konfigurace konečného automatu M , kde $p, q \in Q, a \in \Sigma \cup \{\epsilon\}$ a $x \in \Sigma^*$.

Nechť $r = pa \rightarrow q \in R$ je přechod. Potom M může provést přechod z pax do qx za použití r , zapsáno $pax \vdash qx[r]$ nebo $pax \vdash qx$.

Nechť $\chi_0, \chi_1, \dots, \chi_n$ je běh konečného automatu pro $n \geq 1, \forall i \chi_{i-1} \vdash \chi_i[r_i], r_i \in R$. Pak M provede n přechodů z χ_0 do χ_n , zapsáno $\chi_0 \vdash^n \chi_n[r_1 \dots r_n]$ nebo $\chi_0 \vdash^n \chi_n$. Pokud $\chi_0 \vdash^n \chi_n$ pro nějaké $n \geq 0$, pak $\chi_0 \vdash^* \chi_n$.

Jazyk přijímaný konečným automatem M , je dán jako $L(M) = \{w : w \in \Sigma^*, sw \vdash^* f, f \in F\}$

2.2 Bezkolizní relace

Bezkolizní dva stavy konečného automatu M , jsou takové, které nemohou být aktivní v jednom běhu konečného automatu zároveň. Tato vlastnost je stěžejní pro další zpracování, a to je shlukování do disjunktních množin o počtu prvků n , kde každé dva stavy z této množiny budou mezi sebou bezkolizní, k zakódování do $\log_2 n$ bitového registru.

$R = \{(q_1, q_2) \mid \text{neexistuje slovo } w \text{ takové, že } q_1 \text{ i } q_2 \text{ jsou dosažitelné z některého počátečního stavu konečného automatu přes stejné } w, \text{ kde } w \in \Sigma^*\}$

2.3 Algoritmus 1: Tvorba bezkolizní relace

2.3.1 Popis

Algoritmus 1 simuluje činnost dvou paralelních běhů konečného automatu M přes stejné slovo w . Poté lze rozeznat stavy, které mohou být současně aktivní v rámci jedné konfigurace.

Funguje na bázi grafového algoritmu prohledávání do šířky (při způsobu v jakém pořadí se budou stavy zpracovávat). Jako pomocná struktura je použita fronta, do níž se ukládá dvojice stavů, které spolu tvoří relaci \overline{R} .

Algoritmus 1: Tvorba bezkolizní relace

Vstupy $M = (Q, \Sigma, R, S, F)$ **Výstupy** $R \subseteq Q \times Q$

```
for počáteční_stav_i in S do
  for počáteční_stav_j in S do
    fronta.vlož(počáteční_stav_i, počáteční_stav_j)
    //vloží do fronty dvojici  $S \times S$ 
  end for
end for
while !fronta.prázdná() do
  //algoritmus se provádí dokud není fronta prázdná
  //aktuální_stav bude datového typu dvojice (dva stavy, které mohou být aktivní zároveň)
  aktuální_stav = fronta.získat_prvek()
  for i in přechody_z[aktuální_stav.první].klíče do
    for j in přechody_z[aktuální_stav.druhý].klíče do
      if i == j then
        //z jednoho ze dvou stavů jsme schopni dostat se do jiných dvou přes symbol abecedy
        nový_stav.přidej(přechody_z[aktuální_stav.první][i], přechody_z[aktuální_stav.druhý][j])
      else if j ==  $\epsilon$  then
        nový_stav.přidej(přechody_z[aktuální_stav.druhý][j], aktuální_stav.první)
      else if i ==  $\epsilon$  then
        nový_stav.přidej(přechody_z[aktuální_stav.první][i], aktuální_stav.druhý)
      end if
    end for
  end for
  for stav in nový_stav do
    if stav  $\notin$  Relace then
      Relace.přidej(stav)
      fronta.přidej(stav)
    end if
  end for
end for
end while
```

2.3.2 Popis činnosti

Fronta je při spuštění výpočtu naplněna všemi prvky z $S \times S$ (nesmí být opomenuta varianta (s_1, s_1)), tedy že dvojice stavů jsou stavy totožné). Následně jsou z fronty vyjímány dvojice stavů, algoritmus nahlíží do množiny R (množiny všech přechodových pravidel mezi konfiguracemi) a hledá všechny možné přechody, které konečný automat může provést z následujících dvou konfigurací. Při nalezení tohoto přechodu je simulován přechod do nové konfigurace. Nová dvojice je vložena na konec fronty a uložena do datové struktury, která uchovává hledanou relaci, pokud se v ní ještě tato dvojice nenachází. Výpočet je ukončen, tedy dokončen průchod konečného automatu, až dojde k vyprázdnění fronty.

Zvýšenou pozornost je zde nutno brát na problémové epsilon přechody. Pokud jeden ze zpracovávaných stavů může využít epsilon přechod ke změně konfigurace, je nutné do fronty a zároveň do relace, uložit takto získanou dvojici stavů. Pokud aktuálně zpracovávaný stav má více epsilon přechodů, je nutné je přidat všechny.

2.3.3 Příprava

V prvním kroku je nutné převést testovaná data z formátu *perl compatible regular expressions* (*.pcre), pomocí nástroje frameworku *netbench* [2]. Nástroj převádí do výstupního *timbuk formátu* (*.tmb). Pro lepší manipulaci a přehlednost jsou testovací data převedena do formátu *.vtf.

Samotná příprava algoritmu zahrnuje vytvoření vhodné datové struktury pro uchování přechodových pravidel v konečném automatu. S přihlédnutím na implementaci je použita struktura slovník. Výsledná reprezentace vypadá následovně.

slovník [stav $p \rightarrow$ (slovník [symbol $s \rightarrow$ množina stavu])], kde stav $p \in Q$, symbol $s \in \Sigma$, množina stavů reprezentuje $\in Q$, všechny dostupné stavy dosažitelné přes symbol s ze stavu p .

3 Shlukování stavů

Po vytvoření relace R vyvstává nový problém, a tedy nalezení optimální množiny stavů konečného automatu, které jsou bezkolizní. Tak aby v ideálním případě při lineárně vzrůstajícím počtu stavů, bude vzrůstat logaritmicky o základu dva paměťová složitost tohoto konečného automatu. Cílem této fáze zpracování dat je nalezení takové kombinace množin, aby tvořily, co *nejméně* vzájemně disjunktních množin (množiny budou větší a ušetří více místa).

Při formulaci tohoto problému je zřejmé, že algoritmus musí hledat největší úplné podgrafy grafu (tzv. *kliky*), které budou mezi sebou disjunktní (kde se vrcholem grafu chápe jeden konkrétní stav konečného automatu, a hrana je mezi dvěma vrcholy, pokud jsou dané stavy v relaci R). Graf reprezentuje relace R , popřípadě relace \overline{R} . Není podstatné, zda bude použita relace R nebo její doplněk. Pokud budou prohozeny relace, je nutné pouze přehodnotit podmínky u algoritmu pro hledání úplných podgrafů.

Hledání největšího podgrafu v grafu spadá pod problémy z množiny NP – *úplných* problémů, a tedy není znám algoritmus pro řešení tohoto problému v polynomiálním čase. Lze sestavit algoritmus pro hledání největšího úplného podgrafu v grafu, ovšem pro praktické využití algoritmu musí být vstupní graf velice malý, v tomto případě se musí jednat o konečný automat s malým počtem stavů.

Testovací vstupní data obsahují 140 až 1350 stavů. Tedy je nutné se přiklonit k *heuristickému přístupu* hledání řešení daného problému.

3.1 Algoritmus 2: Hledání úplných podgrafů

3.1.1 Popis

Následující algoritmus sestavuje úplný podgraf z prvního vrcholu, který je uložen v datové struktuře. Výběr dalšího vrcholu, který má být přidán do úplného podgrafu, je ponechán „slepému“ výběru a bere se další vrchol v pořadí (podle použité datové struktury se může výběr měnit). Pokud podgraf s nově přidaným vrcholem stále splňuje podmínku—je úplným podgrafem, pak je přidán. Jinak je přeskočen a zpracovává se další vrchol. Při takto vytvořeném maximálním úplném podgrafu, jsou všechny vrcholy vyňaty z celého grafu a pokračuje se ve hledání dalšího takto maximálního úplného podgrafu.

Výstup algoritmu se bude měnit i s pouhým přeskládáním vrcholů v datové struktuře, nebo pokud bude změněn mechanismus na výběr dalšího vrcholu, který má být přidán do podgrafu. Ovšem výsledky by měly být velice srovnatelné vzhledem k faktu, že síť hran je velice řídká (jak testovací data ukazují). Nemělo by dojít k zásadnímu zdokonalení výsledků.

Algoritmus 2: Hledání úplných podgrafů

Vstupy

vrcholy = seznam všech stavů automatu

\bar{R} = relace viz. výše

X = parametr, do které velikosti budeme kliky hledat, popřípadě jiná ukončovací podmínka

Výstupy

pole_podgrafů = výstup (pole úplných podgrafů, získaných ze vstupu)

Algoritmus**Do**

úplný_podgraf = []

for vrchol **in** vrcholy **do**

 úplný_podgraf = Přidej_vrchol(vrchol, úplný_podgraf)

end for

pole_podgrafů.přidej(úplný_podgraf)

vrcholy -= úplný_podgraf //odstranění stavů, které jsou v úplném podgrafu

While |úplný_podgraf| > X ;

return pole_podgrafů

//funkce

Přidej_vrchol(vrchol, úplný_podgraf)

for vrchol_podgrafu **in** úplný_podgraf **do**

if (vrchol, vrcholy_podgrafu) $\in \bar{R}$ **then**

return úplný_podgraf //nelze vložit do úplného podgrafu

end if

end for

//nenajdeme takovou hranu \Rightarrow lze vložit do úplného podgrafu

úplný_podgraf.přidej(vrchol)

return úplný_podgraf

3.1.2 Popis činnosti

Vstupními parametry tohoto algoritmu jsou v první řadě všechny stavy Q konečného automatu. Dále doplněk relace R , která se získá jako výstup *algoritmu 1*. Následující algoritmus by fungoval, i kdyby byla k dispozici relace R , nikoliv její doplněk. Jedinou změnou v algoritmu by byla negace podmínky pro přidání do úplného podgrafu ve funkci *Přidej_vrchol(vrchol, úplný_podgraf)*. Dalším méně důležitým parametrem je parametr X , který bude ukončovat výpočet. Implicitně je X nastaveno na 0, tzn. budou nalezeny všechny úplné podgrafy i o velikosti 1. Pokud bude X nastaveno na libovolnou kladnou hodnotu. Výpočet bude zastaven, pokud aktuálně získaný podgraf bude mít velikost menší, než je hodnota parametru X .

Výstupem tohoto algoritmu je pole úplných podgrafů, které nejsou nijak seřazeny. Jádro algoritmu je ve funkci *Přidej_vrchol(vrchol, úplný_podgraf)*. Tato část algoritmu se snaží rozšířit aktuální úplný podgraf o další vrchol, který je předán jako jeden z parametrů, druhý parametr je již zmíněný podgraf, který se rozšiřuje. Pokud má být nový vrchol přidán je nutné projít celou relaci R , a nalézt alespoň jednu hranu z každého vrcholu momentálně získaného podgrafu. Pokud je tato podmínka splněna, je vyšetřovaný vrchol přidán do podgrafu. V případě relace \bar{R} , jak ukazuje pseudokód algoritmu výše, je nutné projít celou tuto relaci a nenalezt jedinou hranu, která by spojovala vyšetřovaný vrchol s libovolným vrcholem dočasného podgrafu. Funkce vrátí, ať v případě úspěchu, či neúspěchu, úplný podgraf.

Jakmile algoritmus zkusí přidat všechny stavy konečného automatu a vytvoří maximální, takto získatelný

úplný podgraf, je tento podgraf odstraněn z celého grafu konečného automatu. Tedy v následujícím kroku algoritmus hledá největší možný úplný podgraf ze zbylých vrcholů. Za podmínky, že platil vztah:

$$|podgraf| > X$$

, lze předpokládat, že další takto nalezený graf bude velikostně menší než dříve nalezený. Ovšem toto nemusí nutně platit, bude především záležet na výběru vrcholů, které má daný podgraf obsahovat.

3.1.3 Příprava

Pro správný běh algoritmu je nutné sestavit relaci R , popřípadě její doplněk (lze získat použitím *algoritmu 1*), a vytvoření množiny všech stavů konečného automatu Q . V implementaci byla množina Q zjištěna, průchodem všech přechodových pravidel mezi konfiguracemi konečného automatu.

Zvolení vhodné hodnoty parametru X , který bude ukončovat výpočet. V praktických testech viz. část experimenty v tomto dokumentu, je použita vždy hodnota 0. Tedy budou nalezeny i podgrafy o velikosti 1. Takovýto výpočet bude ukončen, jakmile se zpracují všechny vrcholy, a tedy množina, ze které se vybírají vrcholy pro rozšíření podgrafu, bude prázdná.

Při implementaci tohoto algoritmu byla zvolena pro reprezentaci relace R / \overline{R} datová struktura: *list* [*tuple*(q_1, q_2), *tuple*(p_1, p_2)] kde $q_1, q_2, p_1, p_2 \in Q$. Jednotlivé přechody jsou uchovány ve dvojici, pokud splňují požadavky relace.

4 VHDL reprezentace konečného automatu

Poslední část dokumentu pojednává o výsledném přetvoření konečného automatu do hardwarové reprezentace na čip FPGA. Budou zde rozebrány dvě možnosti, které budou následně v sekci experimenty, mezi sebou a porovnány vyhodnoceny. Všechny realizace jsou uloženy v git repozitáři.[3]

4.1 Realizace bez kompaktní reprezentace

Pro neupravenou variantu, tedy bez aplikování minimalizace, která je probrána výše. Každý stav Q reprezentuje v konečném automatu *jednabitový registr*. Vstupní symbol je zpracován vhodně sestaveným *dekodérem*.

Kombinační logika, která slouží k rozhodnutí, zda konkrétní stav je v dané konfiguraci konečného automatu aktivní, se skládá z logických AND a OR prvků. V konfiguraci:

```
reg_in_p1 <= (reg_q1 AND symb_decoder(16#61#)) OR (reg_q2 AND symb_sdecoder(16#61#))  
OR '0' ;
```

Kde bude aktivní *maximálně jeden* přechod do tohoto stavu $p1$. Pokud nebude splněna žádná z podmínek pro přechod do stavu $p1$, je nastavena logická hodnota 0 do registru *reg_p1* s následující nástupnou hranou hodinového signálu.

4.2 Realizace s kompaktní reprezentací

Realizace na hardware s použitím minimalizace se liší v několika místech. První stěžejní část jsou registrové paměti, kde na rozdíl od první realizace mohou vhodně vybrané stavy *sdílet* jeden registr (registry už nejsou nutně jednabitové).

Je nutno pozměnit *kombinační logiku*, která musí zakódovat do registru stav, aby bylo jasné, který stav je právě uložen v paměti.

První část logiky bude totožná s variantou, kde není aplikována minimalizace. Tedy logické AND a OR prvky s dekodérem pro vstupní symboly. Logika funguje naprosto stejně až na výjimku, že hodnota na *reg_p1* nebude s následujícím taktem hodinového signálu uložena do registru.

Takto získané hodnoty budou seřazeny do správného pořadí a zarovnány na nejbližší vyšší mocninu čísla 2. To znamená, že je vytvořen jeden více bitový signál, který bude zpracovávat *kodeč* 2^n do n , kde n je velikost registru v bitech. Tato technika je použita pro každý více bitový registr. Pouze s různou šířkou vstupu, aby tato šířka odpovídala velikosti registru, podle dříve uvedeného vztahu. Jednabitové registry jsou řešeny stejně jako ve variantě bez minimalizace.

Hodnota zakódovaná kodečem je v následujícím taktu hodinového signálu uložena do příslušného registru. Při změně hodnoty registru je nutné znovu tuto hodnotu rozkódovat. Aby bylo zřejmé, který ze stavů je aktivní. K této činnosti je použit *dekodér* 1 z 2^n , kde pro konkrétní nasazení je n počet bitů registru.

5 Experimenty

Následující praktické výpočty byly prováděny na souborech [4][5], které jsou získány přímo z reálného provozu. Soubory jsou různě komplikované, pro dosažení lepšího spektra výsledků. Nejvíce používané soubory, kde jejich seřazení od nejsložitějších vypadá: *http-backdoor*, *http-malicious*, *http-attacks*.

Výše zmíněná data jsou získána z regulárních výrazů použitých v systému *Snort*. [6]

5.1 Experiment nad Algoritmem 1

$R = \{(q1, q2) | \text{neexistuje slovo } w \text{ takové, že } q1 \text{ i } q2 \text{ jsou dosažitelné z některého počátečního stavu konečného automatu přes } w, \text{ kde } w \in \Sigma^*\}$

Tabulka 1: Alg. 1

Soubor	$ Q $	\bar{R}	R
http-attacks	142	29	10053
http-malicious	249	359	30765
http-backdoor	1358	1656	920424

Z následujících hodnot lze usoudit, že je velice nepraktické počítat relaci R . Při takto velkých vstupech je výstup velice zdlouhavý a v jistém směru zbytečný. Pokud lze velice efektivně spočítat \bar{R} za zlomek času. Potom stačí, pouze za pomoci *doplňku*, přehodnotit výraz, a v dalších algoritmech může být stejně dobře použita relace \bar{R} jako R .

5.2 Experiment nad Algoritmem 2

Tabulka 2: Hledání úplných podgrafů

Soubor	$ Q $	$ \text{úplné grafy} $	$ \text{největší podgraf} $
http-attacks	142	26	89
http-malicious	249	17	85
http-backdoor	1358	46	1234

Sledovaná vlastnost *největší podgraf*, vyjadřuje počet stavů, které tvoří největší dosažený úplný podgraf grafu.

Je zajímavé, že první takto vzniklý úplný podgraf, pokryje zhruba 65% všech stavů konečného automatu. Zde je vidět prostor pro zdokonalení – možnost korekce algoritmu, případně použít jinou heuristiku pro tento problém.

Mírnou výjimku tvoří soubor *http-malicious*, kde dominantní úplný podgraf obsahuje „pouze“ zhruba 35% stavů. Není to jen jeden podgraf, ostatní grafy jsou rozděleny v zajímavém poměru, nejsou zde tvořeny extrémy jako u předchozích dvou vstupů. Další podgraf tvoří 16% a zbývající po 8%. Naproti tomu soubory *http-attacks* a *http-backdoor* mají jeden dominantní graf obsahující 90% stavů, a následující grafy jsou velikosti maximálně 2 (s určitými výjimkami). [7]

Na testovacích datech se s překvapivým výsledkem vytvoří *malý počet* úplných podgrafů s velkou velikostí. Tento stav je velice příznivý.

S tímto spokojivým závěrem, již nebyl implementován další algoritmus na hledání úplných podgrafů. Z důvodů velice dobrých výsledků *algoritmu 2* a předpokládané vysoké časové složitosti následujícího algoritmu.

Dále navržený algoritmus je zdokonalený předcházející algoritmus, v oblasti hledání největšího podgrafu z jednoho konkrétního vrcholu. Toto zdokonalení způsobuje nárůst časové složitosti.

5.3 Experiment nad VHDL reprezentací

Mezi sledované vlastnosti patří nejdelší logická cesta. Z této hodnoty lze snadno vypočítat maximální možnou frekvenci, na které tento obvod může fungovat.

Jelikož se minimalizace zaměřovala na *snížení* počtu registrů, je tedy nutné sledovat hlavně tuto vlastnost. Za cenu snížení počtu registrů se obvod *zkomplikoval*, a zvýšil se počet logických členů. Což je třetí a poslední sledovaná vlastnost obvodu.

Porovnání mezi sebou jsou varianty s minimalizací ku variantám bez minimalizace. Porovnávané hodnoty jsou, jak v absolutních číslech, tak v procentech.

Tabulka 3: Získané údaje z experimentu

regex	Původní reprezentace			Kompaktní reprezentace		
	registry	log. členy	zpoždění	registry	log. členy	zpoždění
Http-attacks	142	158	0.805	42	376	1.464
Http-malicious	224	224	0.409	50	1510	2.138
Http-backdoor	1348	1040	0.445	172	4534	4.946
classification-100g	217	235	0.680	60	2133	2.535
detection-100g	217	236	0.638	33	1182	2.452

Tabulka 4: Porovnání mezi kompaktní a původní realizací

regex	Kompaktní/původní		
	registry	log. členy	zpoždění
Http-attacks	0.2957	2.3797	1.8163
Http-malicious	0.2232	6.7410	5.2274
Http-backdoor	0.1276	4.3586	11.1146
classification-100g	0.2765	9.0766	3.7280
detection-100g	0.1521	5.008	3.8433

U prvního testovaného vstupu *http-attacks* je vidět snížení na 30 % v počtu registrů. Mírně se zvětší i velikost nejdelší cesty oproti dalším regulárním výrazům. A tedy je vidět značné navýšení počtu logických členů na 237 %.

U testovacího souboru *http-malicious* dojde ještě k významnějšímu zmenšení počtu registrů, a to až na 22 % oproti variantě bez minimalizace. Kompaktní konečný automat je již ovšem o dost větší, a proto dojde k velkému navýšení logických členů na 674 %. Velikost nejdelší cesty je více než pětinasobná.

Nejlepší výsledky vykazuje regulární výraz *http-backdoor*, kde je počet registrů snížen až na 12 %. V další sledované vlastnosti – zpoždění obvodu jsou výsledky ze všech sledovaných regulárních výrazů nejhorší.

U souborů *detection-100g* a *classification-100g* je pozoruhodné, že nejdelší cesty jsou mezi sebou shodné, a to jak v původní, tak v kompaktní variantě. Ovšem hodnoty logických členů a počty registrů po minimalizaci se velice liší, regulární výraz *detection-100g* dosahuje mnohem lepších výsledků.

6 Závěr

Cílem této práce bylo prozkoumat některé možnosti minimalizace konečných automatů pro inspekci síťového provozu. V práci se teoreticky vysvětlují použité algoritmy a následně i jejich praktické nasazení na testovacích datech. Samotná minimalizace proběhla ve více krocích.

Překvapivé závěry jsou utvořeny z výsledků prvního algoritmu, kde je hledaná relace velice nevyvážená a řídká. Obsahuje velmi malý počet prvků, a tedy je velice výhodné pokračovat v minimalizaci s doplňkem této množiny. Viz sekce s experimentem nad prvním algoritmem.

Významné výsledky vykazuje i druhý algoritmus při hledání úplných podgrafů v grafu. Sice velice záleží na upřádní vstupní množiny, ale výsledky ukazují, že je možné v některých případech dostat do jednoho úplného podgrafu až 90 % všech stavů konečného automatu. Zde je prostor pro zlepšení algoritmu, avšak velice pravděpodobně za zvýšení časové složitosti (potýká se s *NP – úplným problémem* — algoritmus je *heuristické řešení*).

Cílová minimalizace byla porovnána s neminimalizovaným konečným automatem na hardwarové úrovni pomocí jazyka *VHDL*. Srovnáním vstupních testovacích dat jsem došel k závěru, že výsledná minimalizace proběhla úspěšně v parametru počtu registrů. Výsledné řešení se dostává na 10 – 30 % z původního počtu registrů. Dále existuje vztah mezi registry a nejdelší možnou cestou. Pokud se výsledná minimalizace dostane zhruba na 30 %, výsledná nejdelší cesta není o tolik prodloužená (tzn. čím úspěšnější bude kompakce konečného automatu, tím větší bude nejdelší cesta v obvodu).

Cena za menší počet registrů je ovšem prodloužení nejdelší cesty a značná logická komplikace obvodu, a to v počtu a uspořádání logických členů. Výsledné řešení se dostává na *několikanásobky* původních počtů. Zde je možnost pro další optimalizace a zdokonalení tohoto řešení.

Bylo nutné naučit se a porozumět velkému množství látky, které je nutné k vytvoření a zpracování této problematiky. Převážná část času byla strávena studiem a následným zamýšlením se nad obecným řešením dané problematiky. Poměrně malou část zabrala samotná implementace a korekce chyb.

Reference

- [1] ČEŠKA, M; HAVLENA, V; HOLÍK, L; LENGÁL, O; VOJNAR, T.: Approximate Reduction of Finite Automata for High-Speed Network Intrusion Detection, URL: <https://arxiv.org/pdf/1710.08647.pdf>
- [2] Webové stránky nástroje Netbench, <http://merlin.fit.vutbr.cz/ant/netbench/index.html>
- [3] Git repozitář s vhdl reperezentací automatů, https://github.com/MartinDvorak/IP1/tree/master/vhdl_code
- [4] Git repozitář se zdrojovými regulárními výrazy, <https://github.com/ondrik/appreal/tree/e2e7cb0c7abb7ddccdb7e023127a547e0c550842/experiments/tacas18/regex>
- [5] Git repozitář se zdrojovými regulárními výrazy, <https://github.com/ondrik/appreal/tree/master/regexps/home-brewed>
- [6] Webové stránky nástroje Snort, <https://www.snort.org/>
- [7] Git repozitář s výsledky kompaktní minimalizace, https://github.com/MartinDvorak/IP1/tree/master/vtf_result