Vysoké učení technické v Brně
Fakulta informačních technologií

Projektová praxe

Minimalizace automatů pro inspekci síťového provozu

Martin Dvořák

24. ledna 2018

vedoucí: Ing. Ondřej Lengál, Ph.D.

Obsah

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. Kontrola 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 desítky až stovky 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 proto běžný program pro monitorování je schopný správně pracovat. Nejpoužívanější technologií se jeví FPGA. Díky paralelismu, FPGA čipy dovolují 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 určitý paralelismus. V každé paralelní větvi se simuluje činnost jiného konečného automatu. Tím roste plocha. Z tohoto důvodu je redukování velikosti konečných automatů na čipech FPGA velice důležitým úkolem. [1]

Běh konečného automatu je reprezentovaném na FPGA čipu tak že, konkrétní stav se aktivuje (tzn. Nastavení registru), pokud je vstupní signál roven logické jedničce. Tu lze získat kombinační logikou, kde se nachází vstupní symboly a stavy konečného automatu. Pokud je zpracováván vhodný vstupní symbol a správný stav 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 hardwarové vrstvě počtem použitých komponent. V první řadě se jedná o počty registrových pamětí, kde jednobitový registr reprezentuje určitý stav konečného automatu tak, aby obvod mohl stále pracovat na vysokých rychlostech 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 definován jako klasická pětice $M = (Q, \Sigma, R, S, F)$.
Kde Q reprezentuje konečnou množinu všech stavů.
Σ je vstupní abeceda.
R je konečná množina přechodů tvaru: $pa \to q$,kde$p, q \in Q, a \in \Sigma \cup \{\epsilon\}$.
S je množina počátečních stavů, a zároveň je podmnožinou Q.
F je množina koncových stavů, které jsou podmnožinou Q.

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

Nechť pax a qx jsou dvě konfigurace konečného automatu M, kde $p, q \in Q, a \in \Sigma \cup \{\epsilon\} ax \in \Sigma^*$.
Nechť $r = pa \to q \in R$ je přechod. Potom M může provést přechod z pax do qx za použití r, zapsáno $pax| - qx[r]$ nebo $pax| - qx$.

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

Jazyk přijímaný konečným automatem M, $L(M) = \{w : w \in \Sigma^*, sw| -^* f, f \in F\}$

Oproti „klasické“ definici konečného automatu je zde uvažována varianta více počátečních stavů. To znamená, že může být nedeterministicky vybrán počáteční stav z množiny počátečních stavů. Konečný automat je zde chápán, jako částečně deterministický (obsahuje malé procento nedeterministických pravidel). Proto je nutné pracovat s jistou opatrností při návrhu algoritmů, které budou zpracovávat nedeterministický automat.

2.2 Bezkolizní relace

V prvním kroku je nutné vhodně rozdělit stavy konečného automatu do relace, aby bylo možné vytvořit množinu stavů o počtu prvků n, kde každé dva stavy z této množiny budou mezi sebou bezkolizní, k zakódování do log_2 n bitového registru.

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

Implementace takovéto relace je zbytečně komplikované řešení. Oproti tomu není komplikované zjistit, jaké dva stavy mohou být v jedné konfiguraci aktivní zároveň. Proto byl navrhnut algoritmus pro spočítání relace $\overline{R}$ a následně přes doplněk množin dopočítána relace R.

2.3 Algoritmus 1

2.3.1 Popis

Algoritmus 1 simuluje činnost průniku konečného automatu M s konečným automatem M (vytváří průnik nad jedním konečným automatem). Díky tomu 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 niž se ukládá dvojice stavů, které spolu tvoří relaci $\overline{R}$.

```
for počáteční_stav_i in množina_počátečních_stavů do
    for počáteční_stav_j in množina_počátečních_stavů do
        fronta.vlož(počáteční_stav_i,počáteční_stav_j)
        //zdvojeně vloží do fronty všechny kombinace počátečních stavů
    end for
end for
while !fronta.prázdná() do
    //algoritmus provádím dokud mám prvky ke zpracování
    //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])
            end if
            else if j == ϵ then
                nový_stav.přidej(přechody_z[aktuální_stav.druhý][j],aktuální_stav.první)
            end if
            else if i == ϵ then
                nový_stav.přidej(přechody_z[aktuální_stav.první][i],aktuální_stav.druhý)
            end if
        end for
        for stav in nový_stav do
            if stav ∉ 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 možnostmi počátečních stavů (nesmí být opomenuta varianta $(s1, s1)$, 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 takového to 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 $\in Q$, symbol $\in \Sigma$, množina stavů reprezentuje $\in Q$, všechny dostupné stavy dosažitelné přes symbol s ze stavu p.

2.3.4 Závěr

Podrobný závěr uveden v kapitole experimenty. Za zopakování stojí vlastnost, že je komplikované a *časově náročnější* počítat relaci R. Při dalším zpracování dat je dostačující použití relace $\overline{R}$ a matematické negace výrazů.

3 Shlukování stavů

Po vytvoření relace R vyvstává nový problém, a tedy nalezení optimální množiny stavů konečného automatu, které jsou bezkolizní. Při lineárně vzrůstajícím počtu stavů, bude logaritmicky o základu dva vzrůstat 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 podgrafy grafu, které budou mezi sebou disjunktní (kde se vrcholem grafu chápe jeden konkrétní stav konečného automatu, a hrana mezi dvěma vrcholy, pokud jsou v relaci R). Graf reprezentuje relace R, popřípadě relace $\overline{R}$. Není podstatné, zda bude použita relace R nebo její doplněk.

Hledání největšího podgrafu v grafu spadá pod problémy z množiny *NP – úplných* problémů, a tedy není znám algoritmus pro tohoto problému v polynomiálním čase. Lze sestrojit 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. Při nabytí informace, že získaný graf ze vstupního automatu, kde hrany jsou relační spojení R, má vrcholy velice silně propojeny. Pokud za relační spojení bude uvažována $\overline{R}$, jak je prezentováno v následujícím algoritmu, bude graf velice řídký, co se týče počtu hran.

3.1 Algoritmus 2

3.1.1 Popis

Na základě informace zjištěné z předchozího odstavce, 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 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 algoritmu.

Vstupy
vrcholy = seznam všech stavů automatu
$\overline{\text{R}}$ = 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_podgrafů)* $\in \overline{R}$ **then**
return *úplný_podgraf* //nelze vložit do úplného podgrafu
end if
end for
//nenajdeme takovou hranu =¿ 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 dopočítaný podgraf bude mít velikost menší, než je hodnota parametru X.

Výstupem tohoto algoritmu je pole úplných podgrafů, které jsou pseudo-seřazeny podle velikosti, kdy je první podgraf s největší velikostí (velikost je chápána jako počet vrcholů podgrafu). 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 jíž 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 $\overline{R}$, jak ukazuje pseudokód algoritmu výše, je nutné projít celou tuto relaci a nenalézt jedinou hranu, která by spojovala vyšetřovaný vrchol s libovolným vrcholem dočasného podgrafu. Funkce vrací, 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. Z obecnějšího hlediska se velikost bude zmenšovat.

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, projitím 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(q1, q2), tuple(p1,p2)] Kde $q1, q2, p1, p2 \in Q$. Jednotlivé přechody jsou uchovány ve dvojici, pokud splňují požadavky relace.

3.1.4 Závěr

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

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. Zdokonalení způsobuje nárůst časové složitosti.

4 VHDL reprezentace konečného automatu

Poslední část dokumentu pojednává o výsledném přetvoření konečného automatu do hardwarové reprezentace na FPGA čip. Budou zde rozebrány dvě možnosti a následně v sekci experimenty, budou porovnány mezi sebou a vyhodnoceny.

4.1 Realizace bez minimalizace

Pro neupravenou variantu, tedy bez aplikování minimalizace, která je probrána výše. Každý stav Q reprezentuje v konečném automatu *jednobitový 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. Ve formátu:
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. [3]

4.2 Realizace s minimalizací

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 jednobitové).

Je nutno pozměnit *kombinační logiku*, která musí zakódovat do registru stav, aby bylo jasně poznatelné, 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 uložená 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 *kodér* 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. Jednobitové registry jsou řešeny stejně jako ve variantě bez minimalizace.

Hodnota zakódovaná kodérem je v následujícím taktu hodinového signálu uložena do příslušného registru. Při změně hodnoty registru je nutné znovu tuto hodnotu rozkódovat. Aby bylo naprosto jasné, 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 třech stěžejních souborech [4][5], které jsou získány přímo z reálného provozu. Soubory jsou různě komplikované, pro dosažení lepšího spektra výsledků, kde seřazení od nejsložitějších vypadá: *http-backdoor, http-malicious, http-attacks*.

Výše zmíněná data jsou získaná z regulárních výrazů pomocí systému *Snort*. [6]

5.1 Experiment nad Algoritmem 1

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

Tabulka 1: Alg. 1

Soubor	$\lvert Q\rvert$	$\overline{\mathrm{R}}$	*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 $\overline{R}$ za zlomek času. Potom stačí, pouze za pomoci *negace*, přehodnotit výraz, a v dalších algoritmech může být stejně dobře použita relace $\overline{R}$ jako R.

5.2 Experiment nad Algoritmem 2

Tabulka 2: Alg 2.

Soubor	$\lvert Q\rvert$	$\lvert$*úplné grafy*$\rvert$	$\lvert$*největší(úplné grafy)*$\rvert$
http-attacks	142	26	89
http-malicious	249	17	85
http-backdoor	1358	46	1234

Je zajímavé, že první takto vzniklý úplný podgraf, pokryje zhruba 65% všech stavů konečného automatu. Zde je vidět prostor pro zdokonalení – možnost nějaké korekce, případně vytvoření drobného algoritmu, jak zredukovat zbývající stavy do menších úplných podgrafů.

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]

5.3 Experiment nad VHDL reprezentací

Mezi sledované vlastnosti patří nejdelší logická cesta. Z této hodnoty lze snadno vypočítat maximální možná frekvence, na které tento obvod může fungovat. Pomocí vzorce $f = 1/T$, kde T je tato nejdelší cesta v jednotce času.

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

Porovnáni mezi sebou jsou varianty s minimalizaci ku variantám bez minimalizace. Porovnávané hodnoty jsou, jak v absolutních číslech, tak v relativních procentech.

Tabulka 3: Http-attacks

Varianta	*Nejdelší cesta (ns)*	*Počet registrů*	*Počet logických členů*
Bez minimalizace	3.195	142	158
S minimalizací	2.536	42	376

U Prvního testovaného vstupu *http-attacks* je vidět snížení na $30\,\%$ v počtu registrů. Mírně překvapivě se sníží i velikost nejdelší cesty. A tedy je vidět značný navýšení počtu logických členů na $238\,\%$.

Tabulka 4: Http-malicious

Varianta	*Nejdelší cesta (ns)*	*Počet registrů*	*Počet logických členů*
Bez minimalizace	3.591	224	224
S minimalizací	1.862	50	1510

U Testovacího souboru *http-malicous*, dojde ještě k významnějšímu zmenšení počtu registrů, a to až na $22\,\%$ oproti variantě bez minimalizace. Ovšem kódovaný konečný automat je již o dost větší, a proto dojde k velkému navýšení logických členů na $674\,\%$.

Tabulka 5: Http-backdoor

Varianta	*Nejdelší cesta (ns)*	*Počet registrů*	*Počet logických členů*
Bez minimalizace	3.555	1348	1040
S minimalizací	4.946	172	4534

Zde je překvapivé, že dojde k prodloužení nejdelší cesty v obvodu a to o $1.391\,ns$. Počet registrů se zmenší na $13\,\%$ z původního počtu. Avšak počet logických členů naroste na téměř čtyř a půl násobek.

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 upřádání 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 došlel k závěru, že výsledná minimalizace proběhla úspěšně v parametru počtu registrů. Výsledné řešení se dostává na $10 - -30\,\%$ z původního počtu registrů. Registry jsou ovšem více bitové. Výsledná náročnost tedy nebude přesně na tyto procenta snížena.

Maximální cesta v obvodu na čipu se u většiny vstupních data zmenšila. Výjimku tvořil například soubor *http-backdoor*. Tudíž je možné daný obvod ve většině případů taktovat na vyšší frekvence, což je velice příznivé a důležité kritérium.

Cena za menší počet registrů a menší nejdelší cestu v obvodu je ovšem 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 ladění chyb.

Reference

[1] https://arxiv.org/pdf/1710.08647.pdf

[2] http://merlin.fit.vutbr.cz/ant/netbench/index.html

[3] https://github.com/MartinDvorak/IP1/tree/master/vhdl_code

[4] https://github.com/ondrik/appreal/tree/e2e7cb0c7abb7ddccdb7e023127a547e0c550842/experiments/tacas18/regex

[5] https://github.com/ondrik/appreal/tree/master/regexps/home-brewed

[6] https://www.snort.org/

[7] https://github.com/MartinDvorak/IP1/tree/master/vtf_result