**Minimalizace automatů pro inspekci síťového provozu**

TODO – dodělat úvodní stránku

TODO – vygenerovat Obsah

**Úvod**

U monitorování síťového provozu se konečné automaty používají k detekci protokolů, útoků nebo k identifikaci osob. Tyto automaty se získávají pomocí 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, kde plní činnost předfiltrování paketů. Jako vhodná technologie se jeví FPGA.

I přestože se možnosti FPGA čipů stále zvětšují, rychlost páteřních sítí stoupá rychleji. A proto je nutné zavést určitý paralelismus. V každé paralelní větvi se simuluje činnost jiného konečného automatu, a tedy plocha velice narůstá. Proto je redukování velikosti FPGA velice důležité.

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 fungovat na vysokých frekvencích.

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 stav, kdy v dané konfiguraci konečného automatu není aktivní žádný stav z tohoto registru.

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

**Hledání vhodných stavů konečného automatu**

**Konečný automat**

V této práci je definován jako klasická pětice M = (Q, /Sigma, R, S, F).   
Kde Q reprezentuje konečnou množinu všech stavů.   
/Sigma je vstupní abeceda.  
R je konečná množina pravidel tvaru: *pa -> q*, kde p,q je elementem Q, a je elementem /Sigma sjednoceno {/epsilon}.  
S je množina počátečních stavů, a zároveň je podmnožinou Q.  
F je množina koncových stavů, které jsou podmnožinou Q.

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.

**Hledání vhodné relace**

V prvním kroku je nutné vhodně rozdělit stavy konečného automatu do relace, aby bylo možné vytvořit množinu stavů o *n* počtuprvků, kde žádné dva stavy z této množiny nemohou být zároveň v jedné konfiguraci nedeterministického automatu, k zakódování do *log\_{2} n* bitového registru.

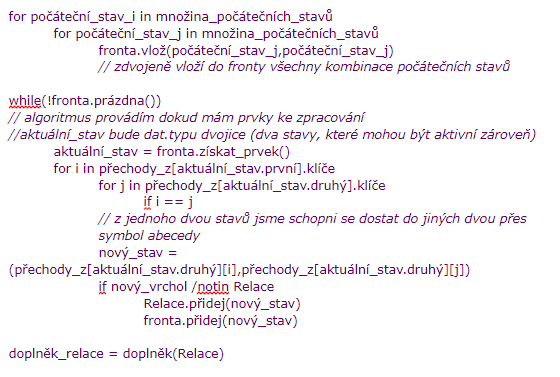
Jako vhodná relace se jeví:   
*R = {(q1, q2) | neexistuje slovo w takové, že q1 i q2 jsou dosažitelné z některého počátečního stavu konečného automatu pres w, kde w je /Sigma^\*}*  
  
Implementace takovéto relace se jeví jako zbytečně komplikované řešení. Oproti tomu není komplikované zjistit, jaké dva stavy mohou být v jedné konfiguraci aktivní zároveň. Proto byl navrhnut algoritmus pro spočítání relace /overline{R} a následně přes matematicky definovaný doplněk množin dopočítání relace R.

**Alg. 1 – hledání relace /overline{R}**

**Popis**

Algoritmus 1 počítá průnik konečného automatu M se sebou samým. 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}.



**Simulace činnosti**

Fronta je při spuštění výpočtu naplněna všemi kombinacemi 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.

**Příprava**

V prvním kroku je nutné převést testovaná data z formátu *perl compatible regular expressions* (\*.pcre). Pomocí nástroje frameworku netbench. Nástroj převádí do výstupního timbuk formátu (\*.tmb). Pro lepší manipulaci a přehlednost jsou testovací data převedena do formátu *valve texture format* (\*.vtf), tento nástroj byl napsán vedoucím této práce.

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 -> (slovník [symbol s -> množina stavu])], kde stav je /in elementem Q, symbol je /in elementem /Sigma, množina stavů reprezentuje /in je elementem Q, všechny dostupné stavy dosažitelné přes symbol s ze stavu p.*

**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.

**Shlukování stavů**

Po vytvoření relace R vyvstává nový problém, a tedy nalezení ideální množiny stavů konečného automatu, které nemohou být aktivní právě v jedné dané konfiguraci zároveň. Při lineárně vzrůstajícím počtu stavů, které splňují tuto podmínku, bude logaritmicky o základu dva vzrůstat paměťová složitost tohoto konečného automatu. Z toho tedy vyplívá, že cílem této fáze zpracování dat je nalezení takové kombinace množin, aby tvořily nejvíce největších vzájemně disjunktních množin.

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 je relační spojení relace 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 nelze najít řešení 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 nasazení takového řešení je z časového hlediska naprosto nereálné.

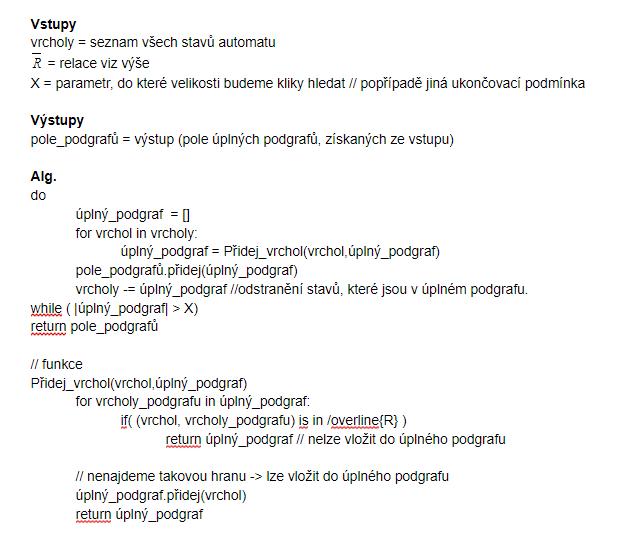
Nutností je přiklonit se k heuristickému přístupu hledání řešení daného problému. Při nabytí informace, že získaný graf ze vstupního automatu, kde hrany jsou relační spojení R, má vrcholy velice silně propojeny. Pokud za relační spojení bude uvažována /overline{R}, jak je prezentováno v následujícím algoritmu, bude graf velice řídký, co se týče počtu hran.

**Algoritmus 2**

**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.



**Simulace 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:   
*|aktuálně získaný 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.

**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 jsou elementem Q. Jednotlivé přechody jsou uchovány ve dvojici, pokud splňují požadavky relace.

**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.

**VHDL reprezentace konečného automatu**

Poslední část dokumentu pojednává o výsledném přetvoření konečného automatu do hardwarové reprezentace.

**Realizace bez minimalizace**

Pro neupravenou variantu, tedy bez aplikování minimalizace, která je probrána výše.   
Každý stav *Q* reprezentujev 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\_decoder(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]

**Realizace s minimalizací**

TODO = doplnit podle toho jak dopadne generování

**Experimenty**

Následující praktické výpočty byly prováděny na třech stěžejních souborech [1], 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. [4]

**Experiment nad algoritmem 1**

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

|  |  |  |  |
| --- | --- | --- | --- |
| **Soubor** | **|Q|** | **/overline{R}** | **R** |
| http-attacks | 142 | 29 | TODO |
| http-malicious | 249 | 359 | TODO |
| http-backdoor | 1 358 | 1 656 | TODO |

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.

**Experiment nad algoritmem 2**

|  |  |  |  |
| --- | --- | --- | --- |
| **Soubor** | **|Q|** | **|úplné grafy|** | **|největší (úplné grafy) |** |
| http-attacks | 142 | 8 | 134 |
| http-malicious | 249 | 6 | 120 |
| http-backdoor | 1 358 | 46 | 1234 |

Je zajímavé, že první takto vzniklý úplný podgraf, pokryje zhruba 90% 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 50% 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). [2]

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.

TODO Cílová minimalizace byla porovnána s neminimalizovaným konečným automatem na hardwarové úrovni pomocí jazyka VHDL.

Vytvoření takovéto práce bylo velice náročné, především časově. 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.

Odkazy

[1] <https://github.com/ondrik/appreal/tree/e2e7cb0c7abb7ddccdb7e023127a547e0c550842/experiments/tacas18/regex>

[2] <https://github.com/MartinDvorak/IP1/tree/master/vtf_result>

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

[4] <https://www.snort.org/>