Tato práce vznikla za podpory projektu TeamIt, jenž je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky prostřednictvím grantu ESF CZ.1.07/2.3.00/09.0067.

BUDOVÁNÍ KONKURENCESCHOPNÝCH VÝZKUMNÝCH TÝMŮ PRO IT

Výzkumná skupina ANT at FIT:

Rychlé hledání regulárních výrazů

Technická zpráva

31.srpna 2010

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ VUT V BRNĚ
Božetěchova 2, 612 66 Brno
tel.: +420 541 141 144, +420 541 212 219, fax: +420 541 141 170, 270
http://www.fit.vutbr.cz, http://teamit.fit.vutbr.cz

Obsah

1 Úvod **2**

2 Regulární výrazy a konečné automaty **3**
2.1 Gramatika . . . 4
2.2 Regulární množiny a regulární výrazy . . . 6
2.3 Konečné automaty . . . 7
2.4 Převod regulárního výrazu na konečný automat . . . 8
2.5 Determinizace automatu . . . 9
2.6 Minimalizace automatu . . . 11

3 Současné architektury pro hledání řetězců a regulárních výrazů **12**
3.1 Zřetězené komparátory . . . 13
3.2 Architektura KMP . . . 13
3.3 Architektura založená na spojování prefixů . . . 14
3.4 Využití paměti ROM . . . 15
3.5 Bloomovy filtry . . . 15
3.6 Využití asociativní paměti . . . 16
3.7 Architektura Bit-Split . . . 17
3.8 Deterministické automaty . . . 18
3.9 Deterministický automat s implicitními přechody . . . 20
3.10 Vlastnosti deterministických automatů . . . 21
3.11 Nedeterministické automaty . . . 22
3.12 Nedeterministický automat se sdíleným dekodérem . . . 23
3.13 Sdílení logiky automatu pro prefixy, infixy a sufixy . . . 24
3.14 Hybridní automaty . . . 25

4 Analýza architektur pro hledání regulárních výrazů **25**

1 Úvod

V posledních letech jsme svědky rychlého rozvoje síťových technologií. Spolu s rozvojem Internetu a síťových služeb ale přichází i rozvoj virů, útoků a jiných bezpečnostních hrozeb. Stále větší důraz je proto kladen na monitorování a bezpečnost počítačových sítí s cílem včas detekovat útok a zajistit účinnou ochranu sítě. Asi nejrozšířenějším prvkem ochrany proti útokům je paketový filtr, který umožňuje podle zadaných pravidel blokovat část síťového provozu. Pravidla jsou ale spojena pouze s informacemi z hlaviček paketů, což neumožňuje detekovat a blokovat některé typy útoku. Proto se pro zajištění bezpečnosti kromě paketových filtrů používají systémy pro detekci nebezpečného provozu (NIDS – Network Intrusion Detection System), které provádí detailní analýzu paketů.

Klíčovou a zároveň nejvíce výpočetně náročnou operací používanou v oblasti monitorování a bezpečnosti počítačových sítí je hledání vzorů v datech paketu. I když byla pro hledání vzorů vyvinuta řada algoritmů [1, 12–14, 28, 29], je možné u NIDS systémů s využitím konvenčních procesorů dosáhnout propustnost v řádu jen několika set Mb/s. Pro detekci útoků je ale potřeba hledat řádově tisíce vzorů na multi-gigabitových rychlostech. Aby bylo zajištěno vyhledávání na takto vysokých rychlostech, používají NIDS systémy různé způsoby hardwarové akcelerace.

Pro urychlení operace hledání vzorů se v komerčních zařízeních [16, 25–27, 36] používají nejčastěji aplikačně specifické integrované obvody (ASIC) nebo programovatelná hradlová pole (FPGA) [49]. Řešení v podobě speciálního integrovaného obvodu umožňuje dosáhnout vysoké frekvence a tím i vysoké propustnosti. Vývoj specializovaného ASIC čipu je ale dlouhodobý a nákladný proces. Stále častěji se proto v oblasti hledání vzorů prosazuje technologie FPGA, která díky svoji programovatelnosti umožňuje přizpůsobit hardwarovou architekturu množině hledaných vzorů a dosáhnout tak vysoké propustnosti srovnatelné se specializovanými ASIC čipy.

Byla proto navržena řada přístupů [3–5, 7, 15, 19, 43, 45–47], kdy je množina řetězců mapována na architekturu vhodnou pro technologii FPGA. Při mapování byly využity jednoduché paralelní komparátory [43] nebo speciálně navržené dekodéry [3, 4, 45], které se snaží efektivně využít hardwarové zdroje na čipu. Další přístupy se snažily umístit množinu vzorů do pamětí mimo čip a redukovat komunikaci s pamětí prostřednictvím vhodně navržených rozptylovacích funkcí [15] nebo Bloomových filtrů [2, 19, 20, 40, 42].

Vern Paxson ukázal [38], že pro identifikaci nebezpečného síťového provozu jsou mnohem účinnější regulární výrazy než řetězce. I když některé navržené přístupy pro hledání řetězců dosahují vysoké propustnosti v řádech i několika desítek gigabitů, nelze je jednoduše rozšířit na hledání regulárních výrazů. Jeden z prvních přístupů umožňující hledat množiny regulárních výrazů vychází z mapování [41] nedeterministického konečného automatu (NFA) do technologie FPGA. Ve výsledné architektuře je možné aktivovat více stavů současně, což umožňuje v každém hodinovém cyklu zpracovat jeden vstupní symbol a zajistit tak lineární časovou složitost vyhledávání. Clark [17, 18] rozšířil mapování o sdílený dekodér, což výrazně redukovalo množství zdrojů potřebných pro reprezentaci nedeterministického automatu. Současně ukázal redukci velikosti nedeterministického automatu s využitím sdílení prefixů. Mapování bylo ještě zdokonaleno o sdílení infixů a sufixů [34], ale tato optimalizace přinesla jen nepatrnou úsporu zdrojů na čipu.

Pro hledání regulárních výrazů byly použity i deterministické konečné automaty (DFA), které umožňují zpracování vstupních dat v lineárním čase. Vlivem determinizace může ale dojít teoreticky až k exponenciálnímu nárůstu počtu stavů automatu, což výrazně zvyšuje paměťové nároky na uložení přechodové tabulky automatu. Yu ukázala [31], že rozdělením automatu na více částí je možné redukovat nárůst počtu stavů způsobený determinizací automatu. Kumar analyzoval vliv struktury regulárního výrazu na velikost deterministického automatu, identifikoval několik klíčových vlastností [30] způsobujících nárůst velikosti automatu a vytvořil architekturu Delay DFA [31], která redukuje velikost automatu zavedením implicitních přechodů. I přes uvedené redukce jsou ale paměťové nároky na reprezentaci deterministického automatu vysoké.

Spojit výhody deterministického a nedeterministického automatu se jako první pokusila Betcchi [8, 10, 11], která vytváří z množiny regulárních výrazů jeden deterministický a několik nedeterministických automatů. Výsledkem je hybridní automat, který se snaží pomocí nedeterministických automatů redukovat nárůst tabulky přechodů automatu deterministického. Časová složitost přijetí jednoho znaku odpovídá počtu aktivních stavů v nedeterministických automatech. U této architektury tak není možné garantovat rychlost vyhledávání, což může využít útočník k zahlcení systému. Další nevýhodou je, že nedeterminis-

tické automaty se nevytvářejí na základě automatu, ale pouze na základě jedné konstrukce používané v regulárních výrazech.

Všechny dosud známé přístupy umožňují hledat regulární výrazy na multi-gigabitových rychlostech jenom za cenu velké paměťové náročnosti nebo za cenu využití velkého množství zdrojů na čipu. U deterministických automatů jsou značné požadavky na velikost a rychlost pamětí, u nedeterministických automatů je potřeba využít velké procento zdrojů na čipu. S využitím vzájemných vazeb mezi modely nedeterministického a deterministického automatu je možné zjistit, že současné způsoby mapování nedeterministického automatu do FPGA nejsou efektivní. V kapitole 4 je ukázáno, že jen malá část stavů nedeterministického automatu může být současně aktivní. To znamená, že jen malá část logiky na čipu se v každém hodinovém cyklu uplatní při výpočtu následujícího stavu. Je proto důležité hledat nové algoritmy a architektury pro hledání regulárních výrazů s cílem dosáhnout co největší propustnost za cenu malého množství hardwarových prostředků.

Technický report je členěna celkem do čtyř kapitol. Na úvod navazují základní definice, věty a algoritmy z oblasti formálních jazyků a automatů, které jsou využívány v dalších částech technického reportu. Ve třetí kapitole jsou představeny dosud publikované architektury pro hledání vzorů a uvedeny jejich vlastnosti z pohledu rychlosti, paměťové náročnosti a možnosti uplatnění pro hledání regulárních výrazů. V navazující čtvrté kapitole je provedena analýza vlastností deterministických a nedeterministických automatů při hledání regulárních výrazů. Analýza uvádí problémy obou automatů a ukazuje, že v nedeterministickém automatu není nutné řešit nedeterminismus na úrovni všech stavů automatu, jak je tomu u dosud publikovaných přístupů.

2 Regulární výrazy a konečné automaty

Aby bylo možné provést srovnání a analýzu současných přístupů pro hledání regulárních výrazů je potřeba si nejprve definovat základní pojmy spojené s řetězci, jazyky, regulárními výrazy a konečnými automaty. Jednotlivé definice, věty a algoritmy z teorie formálních jazyků a automatů jsou převzaty z [24,33,48,50], kde je možné najít také důkazy ke zde uvedeným tvrzením.

Definice 2.1 (Abeceda)**.** *Abeceda je neprázdná množina prvků, které se nazývají symboly abecedy.*

Definice 2.2 (Řetězec)**.** *Řetězcem (také slovem nebo větou) nad danou abecedou se nazývá každá konečná posloupnost symbolů abecedy. Prázdná posloupnost symbolů, tj. posloupnost, která neobsahuje žádný symbol, se nazývá prázdný řetězec a značí se symbolem ε.*

Pro popis abecedy, symbolů nebo řetězců bude v dalším textu použito značení podle následující konvence:

- Velká řecká písmena Σ budou značit abecedy.
- Malá latinská písmena $a, b, c, d, f, ...$ budou značit symboly.
- Malá latinská písmena $t, u, v, w, x, ...$ budou značit řetězce.

Definice 2.3 (Konkatenace řetězců)**.** *Nechť u a v jsou řetězce nad abecedou Σ. Konkatenací (zřetězením) řetězce u s řetězcem v vznikne řetězec uv. Řetězec v je připojen za řetězec u.*

S využitím operace konkatenace je možné definovat podřetězec řetězce, prefix a sufix:

Definice 2.4 (Podřetězec, prefix, infix, sufix)**.** *Nechť w je řetězec nad abecedou Σ. Řetězec y se nazývá podřetězcem řetězce w, pokud existují řetězce x a z takové, že $w = xyz$. Pokud $x = \varepsilon$, nazývá se y prefix řetězce w. Pokud $z = \varepsilon$, nazývá se y sufix řetězce w. Pokud $x \neq \varepsilon$ a $z \neq \varepsilon$, nazývá se y infix řetězce w.*

Definice 2.5 (Délka řetězce)**.** *Délka řetězce je nezáporné celé číslo udávající počet symbolů řetězce. Délka řetězce w se symoblicky značí $|w|$. Je-li $w = a_1a_2...a_n$, $a_i \in \Sigma$ pro $i = 1, 2, ...n$, pak $|w| = n$. Délka prázdného řetězce je nulová, tj. $|\varepsilon| = 0$.*

Definice 2.6 (Jazyk nad abecedou Σ)**.** *Nechť Σ je abeceda. Symbol Σ^* značí množinu všech řetězců nad abecedou Σ včetně řetězce prázdného a symbol Σ^+ značí všechny řetězce nad abecedou Σ vyjma řetězce prázdného, takže platí $\Sigma^* = \Sigma^+ \cup \{\varepsilon\}$. Množina L, pro kterou platí $L \subseteq \Sigma^*$ (případně $L \subseteq \Sigma^+$, pokud $\varepsilon \notin L$), se nazývá jazykem L nad abecedou Σ.*

Protože jazyk je definován jako množina řetězců, je možné s jazyky provádět obvyklé množinové operace jako je *sjednocení*, *průnik*, *rozdíl* nebo *komplement*. Kromě množinových operací jsou nad jazyky definovány i operace související se skutečnosti, že prvky množin jsou řetězce. Jedná se o operaci *součinu* (konkatenaci) dvou jazyků L_1, L_2 a *iteraci*, případně *pozitivní iteraci* jazyka L.

Definice 2.7 (Operace součinu jazyků)**.** *Nechť L_1 je jazyk nad abecedou Σ_1 a L_2 je jazyk nad abecedou Σ_2. Součinem (konkatenací) jazyků L_1 a L_2 je jazyk $L_1 \cdot L_2$ nad abecedou $\Sigma_1 \cup \Sigma_2$, jenž je definován jako*

$$L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$$

Definice 2.8 (Iterace a pozitivní iterace jazyka)**.** *Nechť L je jazyk nad abecedou Σ. Iterace L^* jazyka L a pozitivní iterace L^+ jazyka L je definována jako*

1. $L^0 = \{\varepsilon\}$
2. $L^n = L \cdot L^{n-1}$, *pro* $n \geq 1$
3. $L^* = \bigcup_{n \geq 0} L^n$
4. $L^+ = \bigcup_{n \geq 1} L^n$

2.1 Gramatika

Gramatika je nejznámější prostředek pro reprezentaci jazyka, který umožňuje pomocí konečné reprezentace definovat konečné i nekonečné jazyky. K popisu jazyka používá dvě vzájemně disjunktní konečné abecedy *nonterminálních symbolů* a *terminálních symbolů*. Současně gramatika definuje množinu přepisovacích pravidel, které umožňují z vyznačeného nonterminálu generovat řetězce tvořené pouze terminálními symboly.

Definice 2.9 (Gramatika)**.** *Gramatika G je čtveřice $G = (N, \Sigma, P, S)$, kde*

1. *N je konečná množina nonterminálních symbolů,*
2. *Σ je konečná množina terminálních symbolů, $N \cap \Sigma = \emptyset$,*
3. *P je konečná podmnožina kartézského součinu*

$$(N \cup \Sigma)^* N (N \cup \Sigma)^* \times (N \cup \Sigma)^*$$

4. *$S \in N$ je výchozí nebo také počáteční symbol gramatiky.*

Prvky (α, β) množiny P se nazývají přepisovací pravidla a značí se $\alpha \to \beta$. Řetězce α resp. β se nazývají levou resp. pravou stranou přepisovacího pravidla.

Aplikací přepisovacích pravidel je možné z výchozího symbolu gramatiky odvodit řetězce terminálních symbolů. Aplikaci jednoho pravidla popisuje relace přímé derivace. Tranzitivní a reflexivní uzávěr relace přímé derivace pak umožňuje definovat jazyk generovaný gramatikou.

Definice 2.10 (Relace derivace)**.** *Nechť $G = (N, \Sigma, P, S)$ je gramatika a nechť $\lambda, \mu \in (N \cup \Sigma)^*$. Mezi řetězci λ a μ platí relace $\Rightarrow$ nazývaná přímá derivace, jestliže můžeme řetězec λ a μ vyjádřit ve tvaru:*

$$\lambda = \gamma \alpha \delta$$

$$\mu = \gamma\beta\delta$$

kde $\gamma, \delta \in (N \cup \Sigma)^*$ *a* $\alpha \rightarrow \beta$ *je nějaké přepisovací pravidlo z* P. *Relace přímé derivace mezi řetězci* λ *a* μ *se značí* $\lambda \Rightarrow \mu$. *Tranzitivní a reflexivní uzávěr relace přímé derivace se nazývá derivace a značí se* $\Rightarrow^*$.

Definice 2.11 (Jazyk generovaný gramatikou)**.** *Nechť* $G = (N, \Sigma, P, S)$ *je gramatika. Jazyk* $L(G)$, *který je generován gramatikou* G, *je definován jako množina řetězců:*

$$L(G) = \{w | S \Rightarrow^* w \wedge w \in \Sigma^*\}$$

Jazyky generované gramatikou je možné rozdělit do několika typů podle Chomského klasifikace jazyků. Chomského hierarchie jazyků (nebo také gramatik) vymezuje čtyři typy gramatik podle tvaru přepisovacích pravidel, které jsou obsaženy v množině přepisovacích pravidel P. Tyto typy se označují jako typ 0, typ 1, typ 2 a typ 3.

Typ 0 – obsahuje pravidla v nejobecnějším tvaru, který je shodný s definicí 2.9.

$$\alpha \rightarrow \beta, \quad \alpha \in (N \cup \Sigma)^* N (N \cup \Sigma)^*, \quad \beta \in (N \cup \Sigma)^*$$

Gramatiky typu 0 se nazývají také *gramatikami neomezenými*.

Typ 1 – obsahuje pravidla tvaru:

$$\alpha A \beta \rightarrow \alpha \gamma \beta, \quad A \in N, \quad \alpha, \beta \in (N \cup \Sigma)^*, \quad \gamma \in (N \cup \Sigma)^+, \quad \text{nebo} \quad S \rightarrow \varepsilon$$

Gramatiky typu 1 se nazývají také *gramatikami kontextovými*, neboť tvar pravidla této gramatiky implikuje, že nonterminál A může být nahrazen řetězcem γ pouze tehdy, je-li jeho levým kontextem řetězec α a pravým kontextem řetězec β.

Typ 2 – obsahuje pravidla tvaru:

$$A \rightarrow \gamma, \quad A \in N, \quad \gamma \in (N \cup \Sigma)^*$$

Gramatiky typu 2 se nazývají také *gramatikami bezkontextovými*, neboť substituci pravé strany pravidla γ za nonterminál A je možné provést bez ohledu na kontext, ve kterém se nonterminál nachází.

Typ 3 – obsahuje pravidla tvaru:

$$A \rightarrow xB \text{ nebo } A \rightarrow a, \quad A, B \in N, \quad x \in \Sigma^* \quad \text{nebo} \quad S \rightarrow \varepsilon$$

Gramatika s tímto tvarem pravidel se také nazývá *pravá lineární gramatika*, neboť jediný možný nonterminál na pravé straně pravidla je umístěn úplně napravo.

Definice 2.12. *Jazyk generovaný gramatikou typu* i, *kde* $i \in \{0, 1, 2, 3\}$, *se nazývá jazykem typu* i. *Jazyky pak nazýváme neomezené* ($i = 0$), *kontextové* ($i = 1$), *bezkontextové* ($i = 2$) *a regulární* ($i = 3$).

Věta 2.1. *Nechť* L_i *jsou třídy jazyků typu* i, *kde* $i \in \{0, 1, 2, 3\}$. *Pak platí* $L_0 \subset L_1 \subset L_2 \subset L_3$.

V práci se budeme zabývat pouze jazyky typu 3, které je možné popsat speciální pravou lineární gramatikou nazývající se *regulární gramatika*.

Definice 2.13 (Regulární gramatika)**.** *Gramatika $G = (N, \Sigma, P, S)$ se nazývá regulární (pravá regulární), jestliže množina přepisovacích pravidel P obsahuje pravidla pouze ve tvaru $A \rightarrow aB$ nebo $A \rightarrow a$, kde $A, B \in N, a \in \Sigma$. V případě, že $L(G)$ obsahuje prázdný řetězec, pak regulární gramatika obsahuje jediné pravidlo s prázdným řetězcem na pravé straně ve tvaru $S \rightarrow \varepsilon$. Výchozí symbol S se pak nesmí objevit v žádné pravé straně pravidla.*

Věta 2.2. *Třída regulárních jazyků je totožná s třídou jazyků typu 3 Chomského hierarchie.*

Jazyky, které je možné popsat regulární gramatikou, se nazývají *regulární jazyky*. Podle věty 2.2 platí, že třída regulárních jazyků je totožná s třídou jazyků typu 3 Chomského hierarchie.

2.2 Regulární množiny a regulární výrazy

S pojmeme regulární gramatika a regulární jazyk úzce souvisí i pojmy regulární množina a regulární výraz. Nejprve je potřeba definovat pojem regulární množina.

Definice 2.14 (Regulární množina)**.** *Nechť Σ je konečná abeceda. Regulární množina nad abecedou Σ je definována rekurzivně jako*

1. *$\emptyset$ (prázdná množina) je regulární množina nad abecedou Σ*
2. *$\{\varepsilon\}$ (množina obsahující pouze prázdný řetězec) je regulární množina nad abecedou Σ*
3. *$\{a\}$ pro všechna $a \in \Sigma$ je regulární množina nad abecedou Σ*
4. *Jsou li P a Q regulární množiny nad abecedou Σ, pak také $P \cup Q$, $P \cdot Q$ a P^* jsou regulární množiny nad abecedou Σ.*
5. *Množiny, které nelze získat pomocí výše uvedených pravidel, nejsou regulárními množinami.*

Třída jazyků regulárních množin obsahuje $\emptyset$, $\{\varepsilon\}$, $\{a\}$ pro všechny $a \in \Sigma$ a je uzavřena vzhledem k operacím sjednocení, součinu a iteraci. Obvyklou notací pro reprezentaci regulárních množin jsou regulární výrazy.

Definice 2.15 (Regulární výraz)**.** *Regulární výrazy nad abecedou Σ popisují regulární množiny. Jsou definovány rekurzivně jako*

1. *$\emptyset$ je regulární výraz označující regulární množinu $\emptyset$,*
2. *ε je regulární výraz označující regulární množinu $\{\varepsilon\}$,*
3. *a je regulární výraz označující regulární množinu $\{a\}$ pro všechny $a \in \Sigma$,*
4. *Jsou-li p, q regulární výrazy označující regulární množiny P a Q, pak*
   (a) *$(p + q)$ je regulární výraz označující regulární množinu $P \cup Q$,*
   (b) *(pq) je regulární výraz označující regulární množinu $P \cdot Q$,*
   (c) *(p^*) je regulární výraz označující regulární množinu P^*.*
5. *Žádné jiné regulární výrazy nad abecedou Σ neexistují.*

Věta 2.3. *Jazyk L je regulární množinou když a jen když je generovaný gramatikou typu 3.*

Z věty 2.3 plyne, že pomocí regulární množiny a regulárního výrazu je možné reprezentovat libovolný regulární jazyk a současně libovolný regulární jazyk je možné reprezentovat pomocí regulárního výrazu nebo regulární množiny.

2.3 Konečné automaty

Konečný automat je výpočetní model, který pracuje s konečnou množinou stavů, postupně čte vstupní řetězec a přechází mezi stavy podle definovaných přechodů. Formálně je popsán nedeterministický konečný automat pomocí následující definice.

Definice 2.16 (Konečný automat). *Nedeterministický konečný automat je definován 5-ticí $A = (Q, \Sigma, \delta, q_0, F)$, kde*

1. *Q je konečná množina stavů,*
2. *Σ je konečná vstupní abeceda,*
3. *δ je zobrazení $\delta : Q \times \Sigma \rightarrow 2^Q$, kde 2^Q množina podmnožin množiny Q,*
4. *$q_0 \in Q$ je počáteční stav,*
5. *F je množina koncových stavů $F \subseteq Q$.*

Zobrazení δ se nazývá funkcí přechodu. Je-li $\delta : \Sigma \rightarrow Q$, pak se automat A nazývá *deterministický konečný automat*. V případě deterministického automatu předepisuje funkce δ jediný následující stav. U nedeterministického automatu předepisuje funkce δ množinu následujících stavů.

Činnost konečného automatu, je dána posloupností přechodů. Zpracování řetězce začíná v počátečním stavu q_0. Přechody z jednoho stavu do druhého jsou řízeny funkci δ, která na základě aktuálního stavu q_i a právě čteného symbolu $a \in \Sigma$ vstupního řetězce nastavuje automat do stavu q_j. Proces zpracování řetězce konečným automatem popisuje *binární relace přechodů*, která je definovaná nad *konfiguracemi automatu*.

Definice 2.17 (Konfigurace automatu). *Nechť $A = (Q, \Sigma, \delta, q_0, F)$ je konečný automat, pak dvojice $c = (q, w)$, kde $q \in Q$ a $w \in \Sigma^*$, se nazývá konfigurací automatu A. Konfiguraci c nazveme*

- *počáteční konfigurací, pokud $c = (q_0, w)$,*
- *koncovou konfigurací, pokud $c = (q_f, w)$, kde $q_f \in F$.*

Definice 2.18 (Přechod mezi konfiguracemi automatu). *Nechť $A = (Q, \Sigma, \delta, q_0, F)$ je konečný automat. Binární relace přechodu mezi konfiguracemi $(q_i, aw) \vdash (q_j, w)$ je definována pokud $q_j \in \delta(q_i, a)$, kde $q_i, q_j \in Q$ a $w \in \Sigma^*$. Symbol $\vdash^+$ pak značí tranzitivní uzávěr a symbol $\vdash^*$ tranzitivní a reflexivní uzávěr relace přechodů $\vdash$.*

Na základě konfigurace a relace přechodu mezi konfiguracemi je možné definovat přijetí řetězce konečným automatem a jazyk přijímaný konečným automatem.

Definice 2.19 (Přijetí řetězce konečným automatem). *Řekneme, že vstupní řetězec w je přijímán konečným automatem $A = (Q, \Sigma, \delta, q_0, F)$, jestliže $(q_0, w) \vdash^* (q_f, \varepsilon)$, kde $q_f \in F$.*

Definice 2.20 (Jazyk přijímaný konečným automatem). *Jazyk přijímaný konečným automatem A je označován symbolem $L(A)$ a je definován jako množina všech řetězců přijímaných automatem A:*

$$L(M) = \{w | (q_0, w) \vdash^* (q_f, \varepsilon) \wedge q_f \in F\}$$

Věta 2.4. *Třída jazyků, jež jsou přijímány konečnými automaty, je totožná s třídou jazyků typu 3 Chomského hierarchie.*

Z věty 2.4 plyne, že třída jazyků přijímaných konečným automatem odpovídá třídě regulárních jazyků. To znamená, že ke každému regulárnímu výrazu je možné vytvořit odpovídající konečný automat, který přijímá všechny řetězce definované regulárním výrazem.

2.4 Převod regulárního výrazu na konečný automat

Aby bylo možné definovat převod regulárního výrazu na konečný automat, je nejprve potřeba definovat rozšířený konečný automat, který je proti konečnému automatu rozšířen o ε-*přechody*.

Definice 2.21 (Rozšířený konečný automat)**.** *Rozšířený nedeterministický konečný automat je definován 5-ticí* $A = (Q, \Sigma, \delta, q_0, F)$*, kde*

1. Q *je konečná množina stavů,*
2. Σ *je konečná vstupní abeceda,*
3. δ *je zobrazení* $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow 2^Q$*, kde* 2^Q *množina podmnožin množiny* Q*,*
4. $q_0 \in Q$ *je počáteční stav,*
5. F *je množina koncových stavů* $F \subseteq Q$*.*

Pro převod regulárního výrazu popisující regulární množinu R na ekvivalentní rozšířený konečný automat A, pro který platí $L(A) = R$, je možné použít Thomsonův algoritmus.

Algoritmus 2.1 (Thomsonův: Převod regulárního výrazu na rozšířený konečný automat)**.**
Vstup: *Regulární výraz* r *popisující regulární množinu* R *nad abecedou* Σ*.*
Výstup: *Rozšířený konečný automat* A*, pro který platí* $L(A) = R$*.*

1. *Rozložíme regulární výraz* r *na jeho primitivní složky podle rekurzivní definice regulárního výrazu.*
2. (a) *Pro výraz* ε *zkonstruujeme automat na obrázku 1a.*
   (b) *Pro výraz* $a, a \in \Sigma$ *zkonstruujeme automat na obrázku 1b.*
   (c) *Pro výraz* $\emptyset$ *zkonstruujeme automat na obrázku 1c.*
   (d) *Nechť* N_1 *je automat přijímající jazyk specifikovaný regulárním výrazem* r_1 *a nechť* N_2 *je automat přijímající jazyk specifikovaný výrazem* r_2*.*
      i. *Pro výraz* $r_1 \cdot r_2$ *zkonstruujeme automat na obrázku 1d.*
      ii. *Pro výraz* $r_1 + r_2$ *zkonstruujeme automat na obrázku 1e.*
      iii. *Pro výraz* r_1^* *zkonstruujeme automat na obrázku 1f.*

Algoritmus rozkládá regulární výraz na primitivní složky podle rekurzivní definice regulárního výrazu a konstruuje pro ně automaty, které podle operací konkatenace, sjednocení a iterace skládá do jediného automatu.

Výsledkem Thomsonova algoritmu je rozšířený konečný automat, který je někdy také nazývaný nedeterministický konečný automat s ε-přechody. Aby bylo možné převést rozšířený automat na deterministický konečný automat, je nutné definovat funkci ε-*uzávěru*.

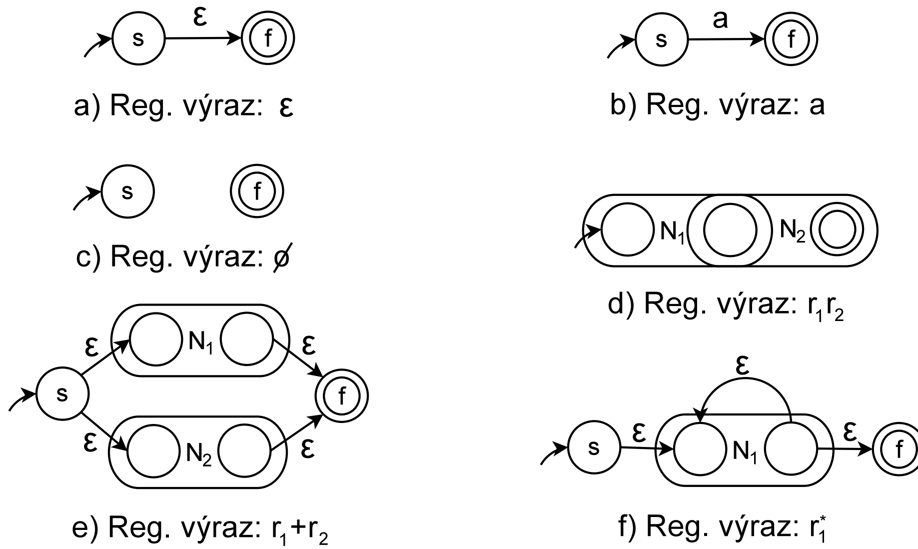
Definice 2.22 (ε-uzávěr)**.** *Funkce* ε*-uzávěr, která k danému stavu určuje množinu všech stavů dostupných po* ε*-přechodech, je definována jako*

$$\varepsilon - uzaver(q) = \{p | \exists w \in \Sigma^* : (q, w) \vdash^* (p, w)\}$$

Pro výpočet funkce ε-uzávěru pro libovolný stav je možné použít následující algoritmus:

Algoritmus 2.2 (Výpočet ε-uzávěru)**.**
Vstup: *Rozšířený konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$*, stav* $p \in Q$*.*
Výstup: ε*-uzávěr*(p)

1. $i := 0$
2. $Q_0 := \{p\}$



Obrázek 1: Konstrukce končeného automatu z regulárních výrazů. Na obrázku jsou konečné automaty pro regulární výrazy (a) ε (b) a (c) $\emptyset$ (d) $r_1 \cdot r_2$ (e) $r_1 + r_2$ (f) r_1^*

3. $i := i + 1$
4. $Q_i := Q_{i-1} \cup \{p | p \in Q \wedge q \in Q_{i-1} \wedge p \in \delta(q, \varepsilon)\ \}$
5. *Pokud* $Q_i \neq Q_{i-1}$ *pak jdi do bodu 3.*
6. ε*-uzávěr*$(p) = Q_i$

Na základě funkce ε-uzávěru je možné definovat algoritmus pro odstranění ε-přechodů z rozšířeného konečného automatu a převést tak rozšířený konečný automat A na ekvivalentní nedeterministický konečný automat A', pro který platí $L(A) = L(A')$.

Algoritmus 2.3 (Odstranění ε-přechodů)**.**
Vstup: *Rozšířený konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$
Výstup: *Nedeterministický konečný automat bez* ε*-přechodů* $A' = (Q, \Sigma, \delta', q_0, F')$*, pro který platí* $L(A) = L(A')$.

1. $\delta' := \emptyset$
2. *Pro všechna* $p \in Q$ *udělej:*

$$\delta' := \delta' \cup \{(p, a) \rightarrow q | a \in \Sigma \wedge q \in Q \wedge p' \in \varepsilon - uzaver(p) \wedge q \in \delta(p', a)\}$$

3. $F' := \{p | p \in Q \wedge \varepsilon - uzaver(p) \cap F \neq \emptyset\}$

Algoritmus nahrazuje ε-přechody tím, že přidává ke stavu všechny přechody ze stavů obsažených v ε-uzávěru. Výsledkem je pak nedeterministický konečný automat, který přijímá stejný jazyk jako rozšířený konečný automat.

2.5 Determinizace automatu

V předchozí kapitole byly definovány algoritmy pro převod regulárního výrazu na ekvivalentní nedeterministický automat. U nedeterministického automatu ale předepisuje přechodová funkce δ množinu následujících stavů, což značně komplikuje implementaci automatu. Je nutné nedeterministicky vybrat

následující stav, což je možné řešit pomocí zpětného vyhledávání (backtrackingu) nebo tzv. Thomsonovou simulací. Při backtrackingu se na základě stavu a vstupního symbolu vybere jeden z možných následujících stavů a pokud neexistuje cesta do koncového stavu, vrací se automat ve vstupních datech zpět. Thomsonova simulace umožňuje aktivovat více stavů současně. V každém kroku se z množiny aktuálních stavů a vstupního symbolu počítá nová množina následujících stavů. Výrazně jednodušší na implementaci jsou deterministické automaty, ve kterých předepisuje funkce δ jediný následující stav.

Následující algoritmus dokáže převést nedeterministický automat A na ekvivalentní deterministický konečný automat A^D, pro který platí $L(A) = L(A^D)$.

Algoritmus 2.4 (Determinizace automatu)**.**

Vstup: *Nedeterministický konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$.

Výstup: *Deterministický konečný automat* $A^D = (Q^D, \Sigma, \delta^D, q_0^D, F^D)$*, pro který platí* $L(A) = L(A^D)$.

Metoda:

1. $Q^D = (2^Q \setminus \{\emptyset\}) \cup \{nedef\}$. *Do* Q^D *je kromě všech podmnožin množiny* Q *přidán stav nedef, do kterého automat přechází v případě, že pro vstupní symbol neexistuje v automatu žádný přechod.*
2. *Polož* $q_0^D = q_0$.
3. *Pro všechna* $S \in 2^Q \setminus \{\emptyset\}$ *a pro všechna* $a \in \Sigma$ *polož:*
   (a) $\delta^D(S, a) = \bigcup_{q \in S} \delta(q, a)$
   (b) *Pokud* $\delta^D(S, a) = \emptyset$*, nastav* $\delta^D(S, a) = nedef$
4. *Polož* $F^D = \{S | S \in 2^Q \wedge S \cap F \neq \emptyset\}$

Algoritmus reprezentuje stavy deterministického automatu jako množiny stavů automatu nedeterministického. Současně deterministický automat obsahuje stav *nedef*, do kterého přechází v případě, že pro vstupní symbol neexistuje v nedeterministickém automatu žádný přechod. Pokud je v nedeterministickém automatu možné přejít na základě vstupního symbolu do množiny stavů, je tato množina v deterministickém automatu chápána jako jeden stav. Tím je zajištěno, že výsledný automat je deterministický, neboť na základě vstupního symbolu je možné přejít nejvýše do jednoho následujícího stavu. Nedeterminismus je ale odstraněn pouze za cenu exponenciálního nárůstu stavů automatu. Výsledný deterministický automat má $2^{|Q|}$ stavů.

Po aplikaci algoritmu mohou být ve výsledném automatu stavy, do kterých nevedou žádné přechody. Takové stavy odpovídají množině stavů, které není možné v nedeterministickém automatu současně aktivovat žádným řetězcem $w \in \Sigma^*$. Pro jejich odstranění je vhodné použít algoritmy v kapitole 2.6.

Na základě algoritmu 2.4 je možné dokázat tvrzení věty 2.5, že libovolný nedeterministický automat je možné převést na ekvivalentní deterministický automat.

Věta 2.5. *Každý nedeterministický konečný automat* A *je možné převést na deterministický konečný automat* A' *tak, že* $L(A) = L(A')$.

Na deterministický automat je možné přímo převést i rozšířený konečný automat vytvořený převodem z regulárního výrazu. Nejprve je ale potřeba zobecnit funkci ε-uzávěru tak, aby mohl být argumentem nejen stav ale i množina stavů $T \subseteq Q$:

$$\varepsilon - uzaver(T) = \bigcup_{s \in T} \varepsilon - uzaver(s) \tag{1}$$

S využitím rozšíření funkce ε-uzávěru na množinu stavů je možné definovat algoritmus přímého převodu rozšířeného konečného automatu na deterministický konečný automat.

Algoritmus 2.5 (Determinizace rozšířeného konečného automatu)**.**

Vstup: *Rozšířený konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$.

Výstup: *Deterministický konečný automat* $A^D = (Q^D, \Sigma, \delta^D, q_0^D, F^D)$*, pro který platí* $L(A) = L(A^D)$.

Metoda:

1. $Q^D = (2^Q \setminus \{\emptyset\}) \cup \{nedef\}$. *Do* Q^D *je kromě všech podmnožin množiny* Q *přidán stav nedef, do kterého automat přechází v případě, že pro vstupní symbol neexistuje v automatu žádný přechod.*

2. $q_0^D = \varepsilon - uzaver(q_0)$.

3. $\delta^D : Q^D \times \Sigma \rightarrow Q^D \cup \{nedef\}$ *je vypočítána takto:*

   - *Nechť* $\forall T \in Q^D, a \in \Sigma : \overline{\delta}(T, a) = \cup_{q \in T} \delta(q, a)$.
   - *Pak pro každé* $T \in Q^D, a \in \Sigma$:

     (a) *Pokud* $\overline{\delta}(T, a) \neq \emptyset$, *pak* $\delta^D(T, a) = \varepsilon - uzaver(\overline{\delta}(T, a))$,

     (b) *jinak* $\delta^D(T, a) = nedef$.

4. $F^D = \{S | S \in Q^D \wedge S \cap F \neq \emptyset\}$.

2.6 Minimalizace automatu

Převod regulárního výrazu na rozšířený konečný automat zavádí poměrně velké množství vnitřních stavů a k velkému nárůstu stavů dochází i při převodu nedeterministického automatu na deterministický automat. Proto obvykle po převodech regulárního výrazu na ekvivalentní deterministický automat následují algoritmy pro minimalizaci automatu. Prvním krokem minimalizace je odstranění *nedosažitelných stavů*, pro které neexistuje posloupnost přechodů z počátečního stavu.

Definice 2.23 (Dosažitelný stav)**.** *Nechť* $A = (Q, \Sigma, \delta, q_0, F)$ *je konečný automat. Řekneme, že stav* $q \in Q$ *je dosažitelný, pokud existuje řetězec* $w \in \Sigma^*$, *pro který platí* $(q_0, w) \vdash^* (q, \varepsilon)$. *Stav je nedosažitelný, pokud není dosažitelný.*

Algoritmus 2.6 (Eliminace nedosažitelných stavů)**.**
Vstup: *Deterministický konečný automat* $A = (Q, \Sigma, \delta, q_0, F)$.
Výstup: *Deterministický konečný automat* A' *bez nedosažitelných stavů.*

1. *Nastav* $i = 0$
2. *Nastav* $S_i = \{q_0\}$
3. *Nastav* $S_{i+1} = S_i \cup \{q | \exists p \in S_i \exists a \in \Sigma : \delta(p, a) = q\}$
4. *Nastav* $i = i + 1$
5. *Pokud* $S_i \neq S_{i+1}$ *jdi do bodu 3.*
6. $A' = (S_i, \Sigma, \delta_{|S_i}, q_0, F \cap S_i)$

Základem algoritmu minimalizace deterministického konečného automatu je koncept nerozlišitelných stavů.

Definice 2.24 (Rozlišení stavů řetězcem)**.** *Nechť* $A = (Q, \Sigma, \delta, q_0, F)$ *je úplně definovaný deterministický konečný automat.*

- *Řekneme, že řetězec* $w \in \Sigma^*$ *rozlišuje stavy* $q_1, q_2 \in Q$, *jestliže* $(q_1, w) \vdash^* (q_3, \varepsilon) \wedge (q_2, w) \vdash^* (q_4, \varepsilon)$ *a právě jeden ze stavů* $q_3, q_4 \in Q$ *patří do množiny* F.

- *Řekneme, že stavy* $q_1, q_2 \in Q$ *jsou* $k-$*nerozlišitelné a píšeme* $q_1 \overset{k}{\equiv} q_2$, *právě když neexistuje* $w \in \Sigma^*$, $|w| \leq k$, *který rozlišuje* q_1 *a* q_2.

- *Stavy* q_1, q_2 *jsou nerozlišitelné, značíme* $q_1 \equiv q_2$, *jsou-li pro každé* $k \geq 0$ $k-$*nerozlišitelné. Relace* $\equiv$ *je ekvivalence na množině stavů* Q.

Věta 2.6. *Nechť $A = (Q, \Sigma, \delta, q_0, F)$ je úplně definovaný deterministický konečný automat a $|Q| = n, n \geq 2$, pak platí:*

$$\forall q_1, q_2 \in Q : q_1 \equiv q_2 \iff q_1 \overset{n-2}{\equiv} q_2$$

S využitím relace nerozlišitelnosti a algoritmu pro odstranění nedosažitelných stavů je možné definovat redukovaný automat a uvést algoritmus pro jeho odvození.

Definice 2.25 (Redukovaný deterministický automat)**.** *Deterministický konečný automat A nazveme redukovaný, jestliže žádný stav $q \in Q$ není nedostupný a žádné dva stavy $q_i, q_j \in Q$ nerozlišitelné.*

Algoritmus 2.7 (Převod na redukovaný deterministický automat)**.**

Vstup: *Úplně definovaný deterministický konečný automat $A = (Q, \Sigma, \delta, q_0, F)$.*

Výstup: *Redukovaný deterministický konečný automat $A' = (Q', \Sigma, \delta', q_0', F')$, pro který platí $L(A) = L(A')$*

1. *Odstraň nedostupné stavy s využitím algoritmu 2.6.*
2. $i = 0$
3. $\overset{0}{\equiv} = \{(p, q) | p \in F \iff q \in F\}$
4. $\overset{i+1}{\equiv} = \{(p, q) | p \overset{i}{\equiv} q \wedge \forall a \in \Sigma : \delta(p, a) \overset{i}{\equiv} \delta(q, a)\}$
5. $i = i + 1$
6. *Pokud* $\overset{i}{\equiv} \neq \overset{i-1}{\equiv}$ *jdi do bodu 4.*
7. $Q' = Q / \overset{i}{\equiv}$
8. $\forall p, q \in Q \forall a \in \Sigma : \delta'([p], a) = [q] \iff \delta(p, a) = q$
9. $q_0' = [q_0]$
10. $F' = \{[q] | q \in F\}$

Algoritmus po odstranění nedosažitelných stavů hledá nad množinou stavů Q relaci nerozlišitelnosti. Využívá skutečnosti, že tato relace je současně ekvivalencí na množině Q a vytváří množinu stavů redukovaného automatu z tříd ekvivalence definovaných rozkladem množiny Q.

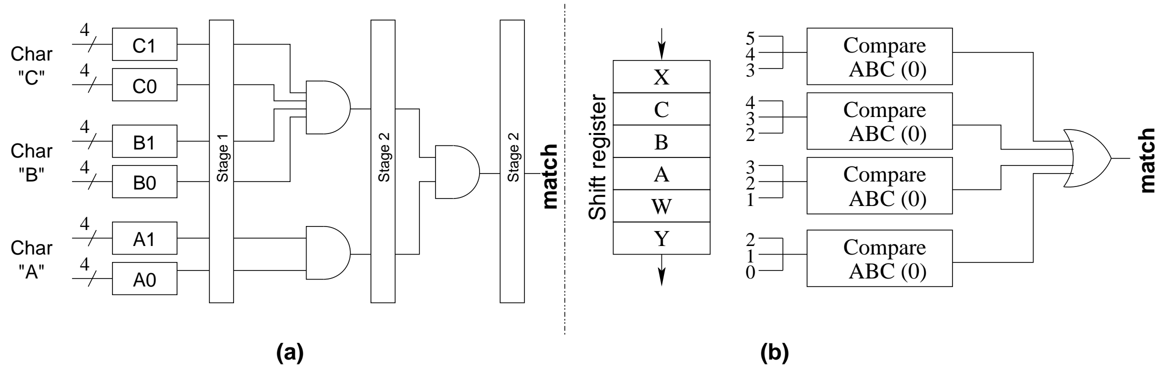
3 Současné architektury pro hledání řetězců a regulárních výrazů

S narůstající propustností síťových linek a zvětšujícím se počtem signatur útoků se současně zvyšují i požadavky na výkonnost systémů pro detekci nebezpečného provozu. Klíčovou a zároveň nejvíc časově náročnou operací je hledání vzorů v datech paketů. I když byla pro hledání vzorů vyvinuta řada algoritmů [1,13,28,29], je možné s využitím konvenčních procesorů dosáhnout propustnost v řádu jen několika set Mb/s. Pro zajištění bezpečnosti na síti je ale potřeba dosáhnout multi-gigabitových rychlostí. Proto v posledních letech vznikla řada hardwarových architektur, které se zabývají urychlením operace hledání řetězců. Byla navržena řada přístupů [3–5, 7, 15, 19, 43, 45–47], u kterých je množina řetězců mapována na hardwarovou architekturu a je dosaženo multi-gigabitových rychlostí.

Pro detekci signatur útoků, virů ale i jiného nebezpečného chování na síti se postupně začaly používat stále víc regulární výrazy, které jsou při identifikaci škodlivého provozu daleko účinnější [38]. I když některé navržené přístupy pro hledání řetězců dosahují vysoké propustnosti, v řádech i několika desítek gigabitů, není možné je rozšířit pro hledání regulárních výrazů. Byly proto navrženy hardwarové architektury zaměřené na hledání nejen řetězců, ale i regulárních výrazů. Publikované přístupy [8,10,11,17,18,30,31,31,41] jsou založeny na modelech deterministického a nedeterministického automatu. Tato kapitola shrnuje charakteristiky dosud známých hardwarových architektur pro vyhledávání řetězců a regulárních výrazů.

3.1 Zřetězené komparátory

Sourdis a Pnevmatikatos ukázali možnost urychlení operace hledání řetězců s využitím technologie FPGA. Vytvořili hardwarovou architekturu [43], která je založena na zřetězených komparátorech. Základem tohoto silně paralelního přístupu jsou čtyřbitové komparátory, hradla `AND` a registry. Schématicky je celá architektura zachycena na obrázku 2a. Architektura je složena z komparátorů, které porovnávají čtveřice bitů vstupních znaků. Vstupní znaky procházejí posuvnými registry, které znaky zpožďují tak, aby bylo možné porovnávat kteroukoliv část hledaného řetězce. Komparátory jsou spojovány do znaků a celých řetězců pomocí operace `AND`. Aby byla dosažena vysoká frekvence, jsou za jednotlivými komparátory i hradly `AND` umístěny registry, které tvoří stupně zřetězené struktury.



Obrázek 2: Architektura založená na zřetězených komparátorech.

Vysoké propustnosti dosahuje architektura nejen díky vysoké frekvenci, ale i zpracováním více bajtů dat v jednom hodinovém cyklu. V takovém případě mohou hledané řetězce začínat na kterémkoliv místě v právě zpracovávaném bloku dat. V architektuře je tento problém řešen replikací komparátorů podle zapojení na obrázku 2b. Na obrázku je vidět, že pro všechny možné posunutí řetězce je vyhrazena jedna sada komparátorů, která zajišťuje porovnání řetězce. Pokud pro některé posunutí najde sada komparátorů hledaný řetězec, nastaví hradlo `OR` příznak úspěšného vyhledání.

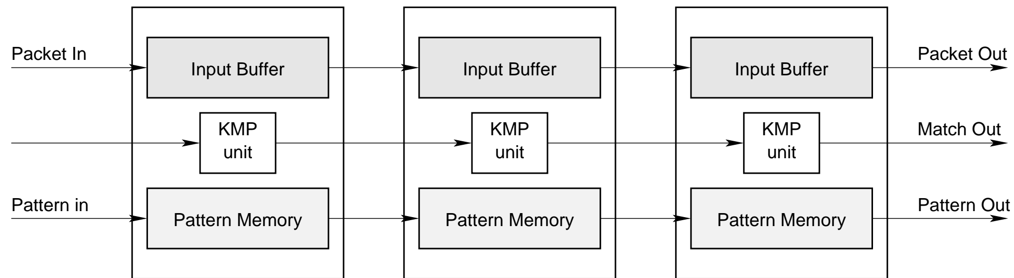
Uvedená architektura je optimalizovaná pro technologii FPGA obsahující look-up tabulky se čtyřmi vstupy. Díky zřetězení jednotlivých operací je možné dosáhnout frekvence až 287 MHz. Ve spojení se zpracováním více bajtů v jednom hodinovém cyklu je tak možné dosáhnout vysoké propustnosti. Na druhou stranu architektura využívá spoustu zdrojů na čipu, takže umožňuje hledat jen velmi malé množiny řetězců.

Aby byly lépe využity zdroje na čipu a bylo možné hledat více řetězců, rozšířili autoři architekturu o sdílený dekodér [44, 45]. Sdílený dekodér převádí znaky na kód 1 z n. Každému znaku odpovídá po překódování jeden signál, který je na základě vstupu nastaven do jedničky nebo do nuly. Posuvné registry zpožďují signály dekódovaných znaků tak, aby přicházely ve správný okamžik na vstup hradel `AND`, které spojují jednotlivé znaky do řetězců. Počet komparátorů je tak redukován na velikost vstupní abecedy, což znamená výraznou úsporu zdrojů na čipu. Díky sdílenému dekodéru bylo dosaženo výrazné zlepšení parametrů, neboť je možné hledat množiny řetězců obsahující nikoliv stovky ale tisíce znaků.

3.2 Architektura KMP

Baker a Prasana publikovali hardwarovou architekturu [5], která je založena na algoritmu KMP [29] a umožňuje hledat množinu řetězců. Algoritmus KMP využívá předem vypočítanou tabulku (π tabulka), kterou se snaží odstranit redundantní porovnání. Díky π tabulce je možné časovou složitost hledání řetězce $O(kn)$ redukovat na $O(k+n)$, kde k je délka hledaného řetězce a n je velikost prohledávaných dat. Architektura založená na algoritmu KMP využívá lineární pole vyhledávacích jednotek. Jejich propojení je zachyceno na obrázku 3.

Každá vyhledávací jednotka se skládá ze tří částí. Obsahuje paměť pro uložení řetězců, jednotku pro realizaci algoritmu KMP a vyrovnávací paměť pro zachycení dat příchozích paketů. Architektura



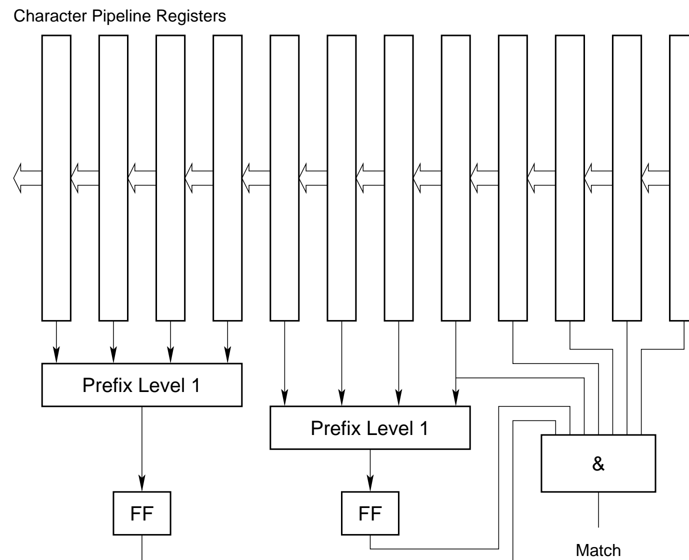
Obrázek 3: Architektura KMP implementovaná v FPGA.

nepotřebuje ke změně množiny hledaných řetězců použít rekonfiguraci FPGA. Hledané řetězce je možné nahrát sériově do jednotlivých jednotek za $p \cdot k$ cyklů, kde p je počet řetězců a k je velikost paměti na řetězce.

Jednotka pro realizaci KMP algoritmu se skládá ze dvou komparátorů, které porovnávají dva po sobě jdoucí znaky vstupních dat. Výsledek druhého komparátoru je uvažován pouze v případě, že bylo první porovnání úspěšné. Tímto způsobem jsou vstupní data zpracovávána rychleji než přicházejí a i v případě neúspěšných porovnání je možné zaručit průměrné zpracování jednoho znaku v jednom hodinovém cyklu. Architektura umožňuje hledání množiny řetězců s řádově tisíci znaky.

3.3 Architektura založená na spojování prefixů

Baker a Prasana navrhli architekturu [3, 4, 6, 7] na obrázku 4, která se snaží pro hledání množiny řetězců efektivně využít zdroje FPGA. Využívá sdílený dekodér ve stejné podobě jako je použit u zřetězených komparátorů. Vstupní znaky jsou dekodérem převáděny na kód 1 z n a každému znaku odpovídá jeden signál, který je podle vstupu nastaven do jedničky nebo do nuly. Dekódované signály pak prochází sérií registrů tvořící zřetězenou strukturu (pipeline), která reprezentuje paměť pro několik posledních znaků.



Obrázek 4: Architektura založená na spojování prefixů.

Signály reprezentující dekódované znaky jsou přiváděny z různých stupňů zřetězené struktury na vstup hradla AND, které spojuje jednotlivé znaky do podřetězců nebo celých řetězců a v případě aktivní úrovně na výstupu indikuje jejich nalezení. Pokud více řetězců obsahuje stejný prefix, je hledání těchto prefixů

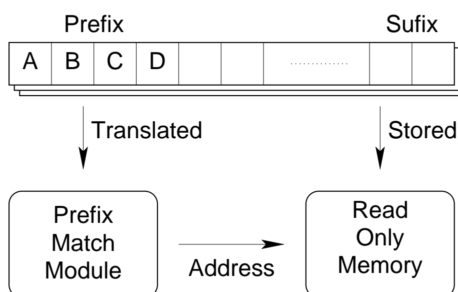
realizováno pouze jednou sadou hradel AND, což výrazně šetří zdroje na čipu.

U tohoto přístupu se používá optimalizace založená na dělení množiny řetězců na menší podmnožiny s cílem redukovat velikost zřetězené struktury. Při dělení je snahou mít v jednotlivých podmnožinách řetězce, které obsahují nejvíce stejných znaků, neboť se tím značně redukuje počet signálů a tím i registrů ve zřetězené struktuře. Podmnožiny jsou vytvářeny algoritmem dělení grafu, jehož uzly reprezentují hledané řetězce a přechody mezi uzly výskyt stejných znaků.

Architektura využívá efektivně zdroje na čipu a umožňuje hledání velkých množin řetězců s řádově tisíci znaky. Současně dokáže pracovat na vysokých frekvencích, což ve spojení s použitím více paralelních jednotek umožňuje dosáhnout multi-gigabitových propustností. Architekturu ale nelze použít pro hledání regulárních výrazů.

3.4 Využití paměti ROM

Mezi další architektury pro vyhledávání řetězců patří přístup založený na porovnání prefixů a následném dohledání zbytku řetězce v paměti ROM [15]. Tento přístup využívá rozdělení množiny řetězců do několika paralelních jednotek. Rozdělení je provedeno tak, aby v rámci jedné jednotky nebyl jeden prefix obsažen v jiném prefixu. V každé jednotce je pak provedeno porovnání vstupních dat vůči množině prefixů. Porovnání je řešeno stejným způsobem, jako v architektuře se sdílenými komparátory. Pokud vstupní řetězec odpovídá některému z prefixů, je v odpovídajícím modulu vygenerována adresa do paměti ROM. Přečte se zbytek řetězce z paměti a provede se porovnání se vstupními daty. Vzor je nalezen pokud skončí porovnání shodou.



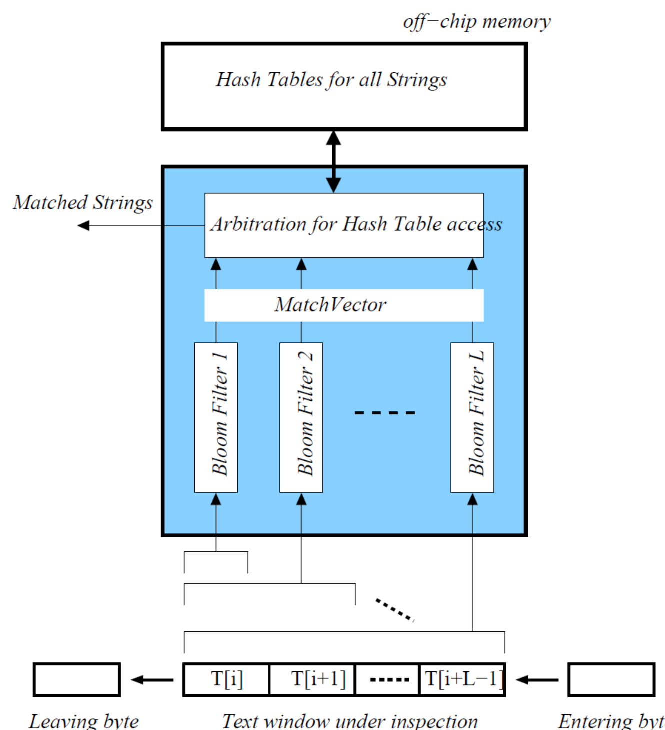
Obrázek 5: Architektura s uložením sufixů v paměti ROM.

Protože jsou jednotlivé sufixy řetězců uloženy v paměti mimo čip, je možné hledat rozsáhlé množiny řetězců. Díky porovnávání prefixů v FPGA je ale ke změně množiny řetězců nutné provést rekonfiguraci. Nevýhodou architektury je, že umožňuje hledat pouze řetězce a nelze ji rozšířit na hledání regulárních výrazů.

3.5 Bloomovy filtry

Dharmapurikar a Lockwood ukázali využití Bloomových filtrů [12, 19, 21] pro rychlé hledání množiny řetězců a aplikovali tento přístup v systémech pro detekci nebezpečného síťového provozu. Autory prezentovaná architektura je založena na skutečnosti, že většina síťových dat neodpovídá žádnému hledanému řetězci. Je tak možné uložit množinu řetězců do relativně pomalé ale velké paměti mimo čip a s využitím Bloomova filtru redukovat počet přístupů do této paměti.

Princip vyhledávací jednotky je zachycena na obrázku 6. Všechny hledané řetězce jsou uloženy v paměti mimo čip na adrese, která odpovídá hodnotě rozptylovací funkce vypočítané z daného řetězce. Vstupní data se posouvají v každém kroku o jeden bajt přes okénko, jehož velikost odpovídá nejdelšímu hledanému řetězci. Nad datovým okénkem pracuje paralelně několik Bloomových filtrů, které odpovídají délkám řetězců z hledané množiny. Cílem Bloomových filtrů je zjistit, jestli ve vstupních datech může být některý z hledaných řetězců dané délky. Pokud ano, přečte se řetězec z externí paměti a provede se porovnání se vstupními daty.



Obrázek 6: Architektura jednotky pro vyhledávání řetězců s využitím paralelních Bloomových filtrů pro redukci přístupu do externí paměti [21].

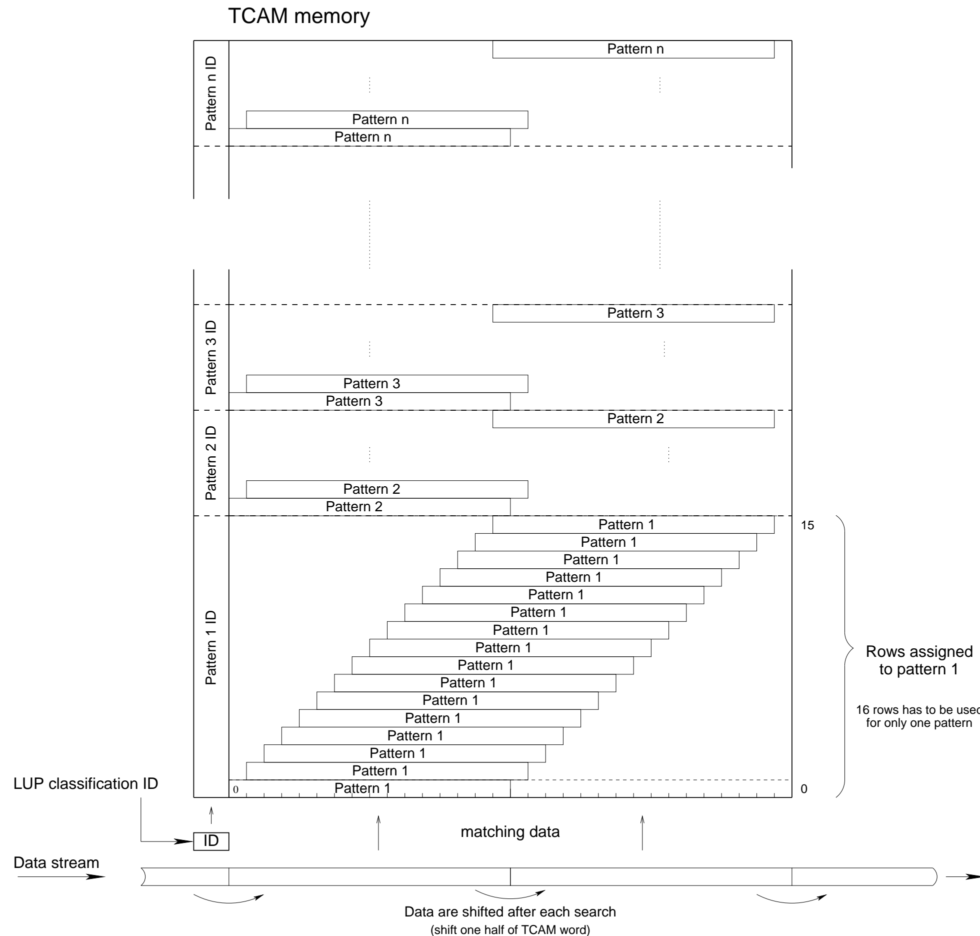
V uvedeném řešení je problém garantovat určitou propustnost. Jednotlivé Bloomovy filtry sice redukují počet přístupů do paměti, ale nelze vyloučit, že bude nutné do externí paměti přistoupit pro všechny délky řetězců. Při hledání velmi krátkých řetězců navíc efektivita Bloomových filtrů velmi silně klesá. Na druhou stranu řetězce jsou spolu s Bloomovy filtry uloženy v paměti, což umožňuje snadnou a rychlou změnu množiny hledaných řetězců bez nutnosti rekonfigurace FPGA.

3.6 Využití asociativní paměti

V rámci evropského projektu Scampi [39], který se zabýval monitorováním sítí na rychlostech 1 Gb/s a 10 Gb/s, byla vyvinuta jednotka [35] pro rychlé vyhledávání řetězců využívající rychlou asociativní paměť. FPGA je v tomto případě využito pouze pro řízení operací, samotné vyhledání je realizováno v asociativní paměti. Aby bylo možné dosáhnout maximální propustnosti, bylo navrženo uspořádání paměti, které je na obrázku 7. Cílem tohoto uspořádání je zejména zpracovat co nejvíce bajtů dat v jednom hodinovém cyklu, což je dosaženo paralelním porovnáním všech řetězců včetně různých posunutí vůči zpracovávanému bloku dat.

Použitá asociativní paměť je konfigurována na šířku slova 272 bitů a kapacitu 8192 slov. Délka vzoru je shora omezena na 16 znaků. Každý vzor je uložen v paměti na 16 řádcích, vždy posunut o jeden bajt proti předchozímu řádku. To umožňuje porovnávat v jednom kroku všechna možná zarovnání řetězců a posouvat vstupní data vždy o polovinu šířky slova asociativní paměti. V jednom vyhledání je tak zpracováno 128 bitů vstupních dat. Jelikož vyhledávací operace trvá čtyři hodinové cykly, je v jednom hodinovém cyklu zpracováno 32 bitů dat. S využitím frekvence 100 MHz tak bylo dosaženo propustnosti 3,2 Gb/s.

Řádově vyšších propustností je možné dosáhnout při použití rychlejší asociativní paměti. Příkladem takové paměti je IDT 75K72100 [37], která umožňuje provést až 125 miliónů vyhledávacích operací pro slovo šířky 576 bitů. S využitím takto výkonné asociativní paměti je možné s uvedeným uspořádáním paměti dosáhnout propustnost 9,6 Gb/s, pro množinu 4096 vyhledávaných řetězců o délce 16 bajtů. Při



Obrázek 7: Uspořádání obsahu asociativní paměti pro vyhledávání množiny řetězců na vysoké rychlosti. Vyhledávané vzory jsou uloženy ve všech možných posunutích vůči vstupnímu toku dat.

využití šířky slova 576 bitů je možné dosáhnout propustnosti 19,2 Gb/s, ale jen za cenu zmenšení velikosti množiny na 1024 řetězců s maximální délkou 16 bajtů.

V porovnání s ostatními architekturami pro hledání řetězců dosahuje navržený přístup při použití asociativní paměti nejlepšího poměru mezi velikostí množiny vyhledávaných řetězců a dosaženou propustností. Výhodou je i možnost prakticky okamžité změny množiny hledaných řetězců bez nutnosti rekonfigurace FPGA. Naopak nevýhodou je zejména omezená délka vyhledávaných vzorů v závislosti na šířce slova asociativní paměti a možnost hledání pouze řetězců.

3.7 Architektura Bit-Split

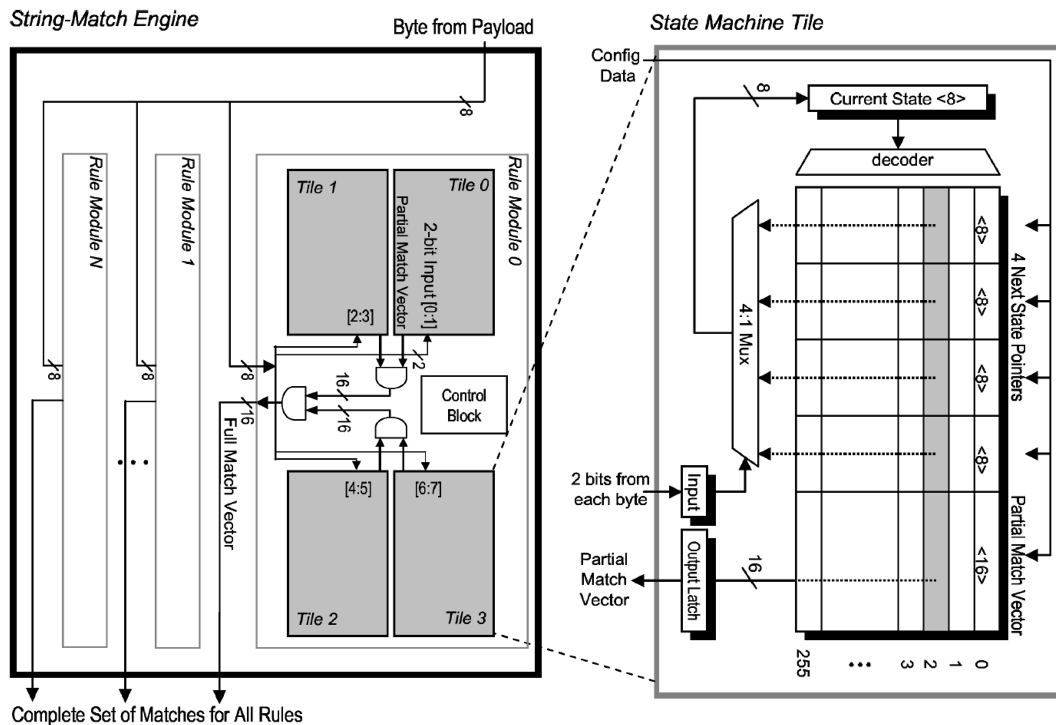
Architektura Bit-Split [46, 47] vychází z algoritmu Aho-Corasickové [1]. Liší se ale způsobem zpracování vstupních dat. Místo jednoho automatu, který zpracovává vstupní data po znacích, je použito několik paralelních automatů, které zpracovávají vstupní data po jednom nebo dvou bitech.

Algoritmus Aho-Corasickové využívá předzpracování množiny vzorů, kdy ještě před samotným vyhledáváním je ve dvou po sobě jdoucích krocích ze zadané množiny vzorů vytvořen deterministický automat. V prvním kroku je vytvořen pro všechny řetězce strom stavů tvořící automat bez zpětných přechodů. V druhém kroku jsou do automatu vloženy zpětné přechody tak, aby byl z každého stavu definován přechod pro všechny vstupní symboly. Protože množiny řetězců pracují nad abecedou 256 znaků, je stupeň větvení automatu roven 256. Ke každému stavu je tak uložen velký počet přechodů do následujících stavů.

Výhodou algoritmu Aho-Corasickové je lineární časová složitost vyhledávání množiny vzorů. Naopak

nevýhodou jsou velké paměťové nároky na uložení všech přechodů. Aby byla snížena paměťová náročnost vyhledávání, využívá architektura Bit-Split místo jediného automatu osm nezávislých automatů. Každý automat je zodpovědný za zpracování jednoho bitu vstupního slova. Řetězec je pak přijat pouze v případě, že všechny automaty se nachází v koncovém stavu. Výhodou rozdělení na automaty zpracovávající jediný bit vstupního slova je, že z každého stavu existují maximálně dva přechody. Výrazně se tak redukuje počet přechodů i složitost výpočtu následujícího stavu. Každý automat tak prochází binární strom se zpětnými hranami.

V [46] je ukázáno, že pro optimální využití hardwarových zdrojů je efektivnější vytvořit místo osmi automatů čtyři automaty, které zpracovávají vstupní data po dvou bitech. Navržena architektura vyhledávací jednotky je zachycena na obrázku 8. Na obrázku je vidět, že jednotka se skládá ze čtyř modulů, které odpovídají jednotlivým automatům. Každý modul obsahuje paměť pro uložení až 256 stavů a umožňuje vyhledávat až 16 dvojic bitů z 16 řetězců.



Obrázek 8: Architektura Bit-Split [46] se skládá z několika paralelních automatů. Každý automat zpracovává jinou část vstupního slova.

S využitím architektury Bit-Split jsou až devět krát sníženy paměťové požadavky proti algoritmu Aho-Corasickové. Je tak možné hledat množiny řetězců obsahující řádově desetitisíce znaků. Protože architektura Bit-Split převádí řetězce na deterministický konečný automat, je možné ji použít nejen k hledání řetězců, ale i pro hledání některých regulárních výrazů. Jsou ale podporovány pouze regulární výrazy, pro které je možné použít i algoritmus Aho-Corasickové [1]. Propustnost architektury BitSplit je omezena propustností použitých pamětí, neboť v každém kroku je potřeba číst velké množství dat. Pro technologii Virtex–5 je možné dosáhnout propustnost až 4 Gb/s na jednu jednotku.

3.8 Deterministické automaty

Deterministický automat je vhodný model k hledání nejen řetězců, ale i regulárních výrazů. S využitím algoritmů 2.1 a 2.5 je možné libovolný regulární výraz převést na ekvivalentní deterministický konečný automat (DFA – Deterministic Finite Automaton). Ke každému stavu a vstupnímu symbolu je pak jednoznačně přiřazen následující stav, což umožňuje snadnou softwarovou, ale i hardwarovou implementaci

Charakteristika reg. výrazu	Příklad	Počet stavů
Řetězec délky k	`^ABCD` `.*ABCD`	$k+1$
Regulární výraz s $.*$	`^AB.*CD` `.*AB.*CD`	$k+1$

Regulární výraz s `^`, `.*` a s omezením na délku j	`^AB.{j+}CD`
	`^AB.{0,j}CD`
	`^AB.{j}CD`
		$O(k*j)$

Regulární výraz s `^`, třídou znaků překrývající prefix výrazu a s omezením na délku j	`^A+[A-Z]{j}D`	$O(k*j^2)$
Regulární výraz s omezením délky j, kde třída znaků nebo `.*` se překrývají s prefixem	`.*AB.{j}CD` `.*A[A-Z]{j+}D`	$O(k*2^j)$

Tabulka 1: Analýza počtu stavů DFA pro různé regulární výrazy délky k.

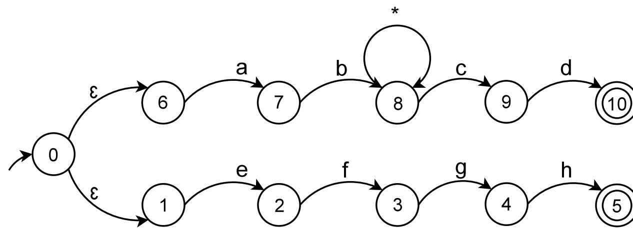
automatu.

Výhodou deterministického automatu je zpracování vstupních dat v lineárním čase. V každém hodinovém cyklu je zpracován jeden vstupní symbol, což znamená konstantní rychlost vyhledávání a tedy i garantovanou propustnost. Nevýhodou deterministického automatu je velká paměťová náročnost. Vlivem determinizace může dojít teoreticky až k exponenciálnímu nárůstu počtu stavů automatu, což způsobuje i exponenciální nárůst velikosti přechodové tabulky. Vzniká tak problém najít dostatečně rychlou a dostatečně velkou paměť pro uložení tabulky přechodů.

Exponenciální nárůst počtu stavů nastává nejen teoreticky, ale teoretické hranici se přibližují i automaty reprezentující používané množiny regulárních výrazů. Nárůst velikosti automatu je většinou způsoben použitím určité konstrukce. Detailní analýzou nárůstu počtu stavů deterministického automatu pro jeden regulární výraz se zabývali Sallesh Kumar a Fang Yu [30, 31]. Velikost DFA pro různé vlastnosti regulárních výrazů délky k shrnuje tabulka 1.

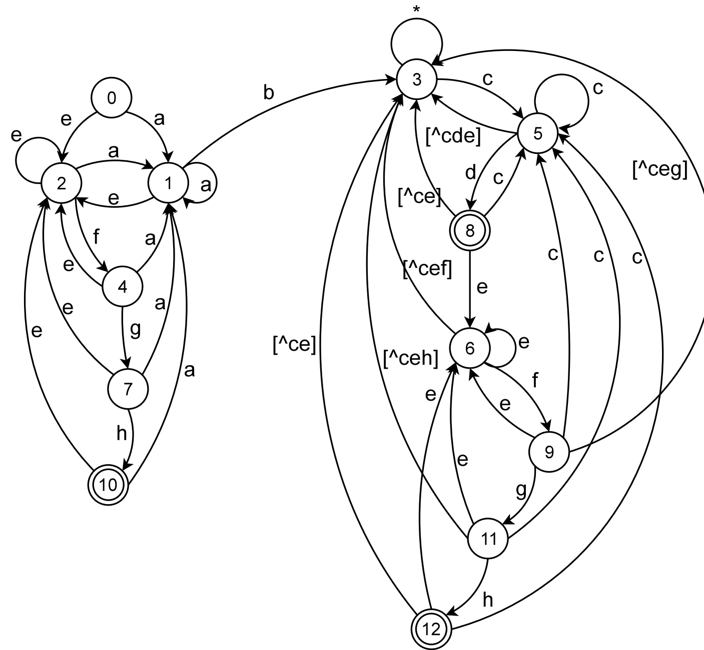
Pro množinu regulárních výrazů se nárůstem počtu stavů zabývala Michela Becchi. V [8] ukázala, že nárůst počtu stavů a přechodů vzniká zejména díky konstrukci $.*$ (tečka-hvězdička) na začátku nebo uprostřed regulárního výrazu. Konstrukce $.*$ popisuje libovolný řetězec abecedy a v deterministickém automatu způsobuje replikaci stavů, které reprezentují jiné regulární výrazy. Výrazný nárůst zaplnění přechodové tabulky automatu způsobují i zpětné přechody, které vznikají v deterministickém automatu, pokud v regulárním výrazu je některý infixový výraz shodný s prefixem.

Nárůst počtu stavů a přechodů je dobře viditelný na příkladu dvou regulárních výrazů $ab.*cd$ a $efgh$. Na obrázku 9 je pro oba výraz vytvořený jeden nedeterministický automat a na obrázku 10 ekvivalentní minimální deterministický automat. Z obrázku je vidět nárůst počtu stavů z 9 na 12 a nárůst počtu přechodů z 12 na 36. Stavy 6, 9, 11 a 12 jsou replikovány ze stavů 2, 4, 7, 10 díky konstrukci $.*$. Současně je vidět i nárůst zpětných přechodů.



Obrázek 9: Nedeterministický automat pro regulární výrazy $ab.*cd$ a $efgh$.

S nárůstem počtu regulárních výrazů přibývá i počet konstrukcí, které způsobují extrémní nárůst



Obrázek 10: Minimální deterministický automat pro regulární výrazy $ab.*cd$ a $efgh$.

velikosti a zaplnění tabulky přechodů. Umístit pak tabulku přechodů automatu do paměti a zajistit dostatečnou rychlost vyhledávání, je velmi obtížné.

3.9 Deterministický automat s implicitními přechody

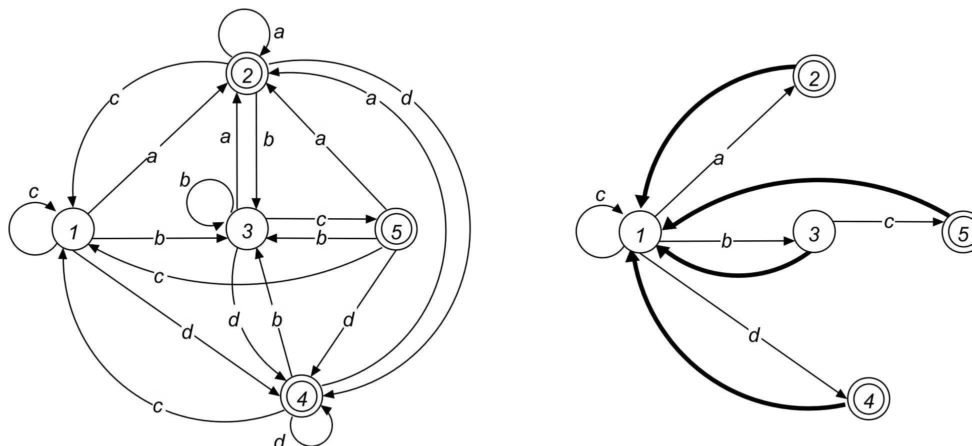
Pro snížení počtu přechodů v přechodové tabulce zavádí Kumar [31] tzv. implicitní přechody. Pokud v aktuálním stavu nelze přijmout vstupní znak, provede se implicitní přechod z daného stavu. Takto upravený automat se nazývá zpožděný automat (Delay DFA).

Delay DFA vychází z pozorování, že v deterministickém automatu $A = (Q, \Sigma, \delta, q_0, F)$ existuje více stavů $q_1, q_2, ...q_k \in Q$, ze kterých vedou přechody $\delta(q_i, a) = q_{dst}$, $i \in \langle 1; k\rangle$ označené stejným symbolem $a \in \Sigma$ do společného třetího stavu $q_{dst} \in Q$. V takovém případě je možné ponechat v automatu z těchto přechodů pouze jeden přechod $\delta(q_i, a) = q_{dst}$, $i \in \langle 1; k\rangle$ a ostatní přechody $\delta(q_j, a) = q_{dst}$, $j \in \langle 1; k\rangle, j \neq i$ nahradit implicitními přechody $\delta(q_j, \varepsilon) = q_i$. Je důležité vhodně vybírat přechody, které budou nahrazeny, neboť implicitní přechody mohou na sebe navazovat a protože nepřijímají žádný znak vstupního řetězce, mohou prodlužovat dobu zpracování vstupních dat. Problém je řešen autory tak, že je omezena maximální délka cesty, která je tvořená implicitními přechody.

Kontrukce Delay DFA z deterministického automatu pro regulární výrazy a^+, b^+c a c^*d^+ je znázorněna na obrázku 11. Na obrázku je u deterministického automatu vidět velký počet zpětných přechodů do stavů 2, 3 a 4. Zavedením implicitních přechodů do stavu 1 je celkový počet přechodů výrazně snížen.

Velkou nevýhodou Delay DFA je snížení propustnosti, neboť implicitní přechody neakceptují žádné znaky, ale potřebují přístupy do paměti. Tento problém se snaží řešit vylepšený algoritmus pro nalezení implicitních přechodů [9]. Autory navržený způsob řešení vychází z pozorování, že zpožděné přechody mohou vést pouze směrem k počátečnímu stavu a nemůže jich být tedy více než je počet dosud přijatých znaků. Pro vytvořený Delay DFA je pak díky konstrukci implicitních přechodů dosaženo časové složitosti $2n$, kde n je počet přijatých znaků. Další optimalizace [32] automatu Delay DFA vychází z ukládání informací o následujícím stavu spolu s přechody, což umožňuje provést zpožděný přechod zároveň s přijetím vstupního znaku.

Další implementace automatu [22] vychází z pozorování, že není třeba mít v rychlé paměti celou přechodovou tabulku, ale pouze aktuální stav spolu s výstupními přechody. Autoři na základě uvedeného



Obrázek 11: DFA a Delay DFA pro regulární výrazy a^+, b^+c a c^*d^+.

pozorování navrhli, aby byla na čipu umístěna rychlá paměť, která obsahuje všechny přechody z aktuálního stavu a s provedením každého přechodu se aktualizovala.

3.10 Vlastnosti deterministických automatů

Sailesh Kumar v [30] analyzoval používané regulární výrazy a vlastnosti z nich vytvořených automatů. Na základě svých pozorování definoval tři základní vlastnosti deterministických konečných automatů pro hledání množin regulárních výrazů:

Insomnia – deterministický konečný automat není schopen reflektovat pravděpodobnost průchodu jednotlivými stavy. Řetězce a regulární výrazy popisují v NIDS systémech různé útoky a jiný nežádoucí provoz. Většina síťového provozu je ale bezpečná a neobsahuje žádné útoky ani jinak nebezpečné pakety. Platí tedy, že s velkou pravděpodobností bude aktivních pouze několik prvních stavů automatu. Ostatní stavy budou aktivovány pouze výjimečně.

Amnesia – deterministický konečný automat má pouze omezenou paměť pro uchování aktuálního stavu vyhledávání. Do této paměti je ukládán pouze aktivní stav automatu bez jakýchkoliv dodatečných informací. Deterministický automat tedy neví, jak vypadal předchozí vstupní řetězec. Tato situace často vede k exponenciálnímu nárůstu stavů při determinizaci, neboť se musí vytvářet všechny možné kombinace stavů. V případě rozšíření stavové informace o několik jednobitových registrů je možno nahradit tyto kombinace nastavením registru a použitím již existujících stavů.

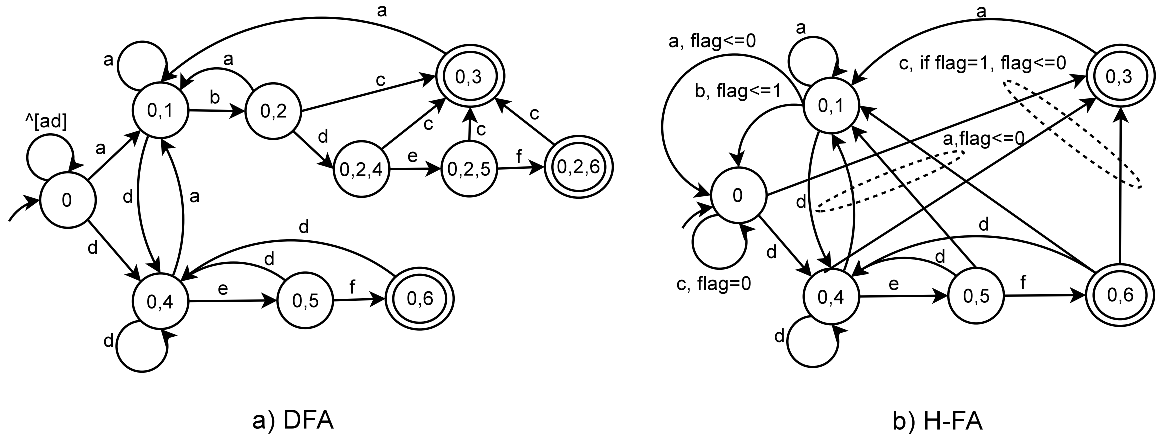
Acalculia – v NIDS systémech se často objevují výrazy, které obsahují pevně stanovený počet opakování. Konečné automaty však nejsou schopny efektivně tyto prvky počítat, a je tedy nutné pro každý z nich vytvářet samostatný stav.

Pro jednotlivé vlastnosti navrhl Kumar [30] také způsoby řešení. Insomnia je řešena pomocí rozdělení konečného automatu na dvě části podle pravděpodobnosti aktivace stavů. Počáteční stav a jeho okolí mají nejvyšší pravděpodobnost aktivace a jsou proto implementovány pomocí rychlého deterministického automatu. Ostatní stavy jsou ponechány ve formě nedeterministických automatů.

Kumarem navržená realizace nedeterministických automatů má tabulku přechodů uloženou v paměti. Protože je pro akceptování příchozího znaku někdy potřeba udělat více přístupů do paměti, je zpracování vstupních dat pomocí NFA pomalejší. Z toho vyplývá, že při přechodu do nedeterministického automatu klesá propustnost vyhledávací jednotky. Nicméně pravděpodobnost aktivace nedeterministického automatu je malá, takže by ke zpomalování nemělo docházet často.

Pro odstranění Amnesie bylo autorem navrženo použít H-FA (History based Finite Automata) [30]. Jedná se o rozšíření konečného automatu o jednobitové registry. S provedením přechodu se registry

nastavují nebo resetují. Současně jsou registry použity k podmínění provedení jiných přechodů. Pro odstranění Acalculia jsou H-FA registry doplněny i o čítače znaků.



Obrázek 12: DFA a H-FA pro regulární výraz $. * ab[^\wedge a] * c|. * def$.

Princip H-FA je vidět na obrázku 12, kde je pro regulární výraz `.*ab[^a]*c|.*def` vytvořen deterministický a H-FA automat. Zatímco v deterministickém automatu jsou replikovány stavy a přechody pro hledání části výrazu `.*def`, v H-FA je jedna ze struktur popisující `.*def` odstraněna pomocí jednobitového registru, který signalizuje přijetí podřetězce `ab`.

3.11 Nedeterministické automaty

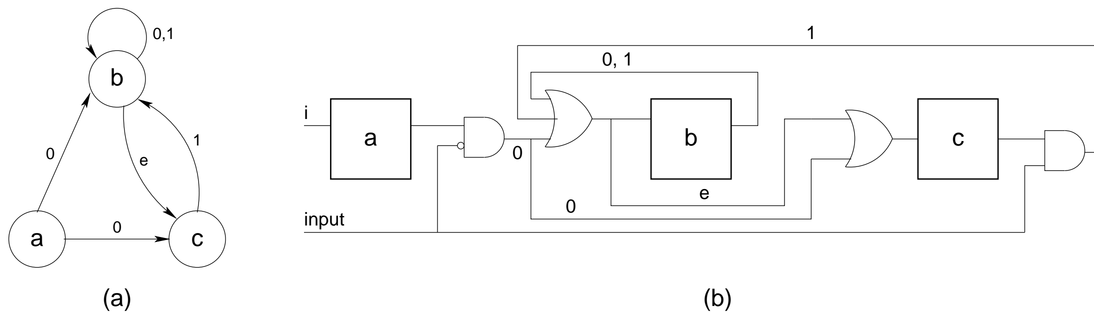
Základním problémem deterministických automatů je paměťová složitost. I přes řadu publikovaných optimalizací je tabulka přechodů deterministického automatu velká. U nedeterministického automatu narůstá počet stavů lineárně s délkou regulárního výrazu. Paměťová složitost pro m regulárních výrazů délky n je $O(nm)$. Velikost paměti pro uložení automatu tak přibližně odpovídá popisu množiny regulárních výrazů. K uložení celé tabulky přechodů je tak potřeba mnohem méně paměti, než v případě deterministického automatu. Na druhou stranu nedeterministické automaty nemají lineární časovou složitost, což může vést ke zhoršení propustnosti. Proto architektury na bázi nedeterministických konečných automatů vyžadují pro garantování propustnosti vysoký stupeň paralelního zpracování.

První architektura založená na bázi nedeterministických automatů [41] byla motivována rychlou konstrukcí rozšířeného konečného automatu pomocí Thomsonova algoritmu a přímým mapováním automatu do technologie FPGA. Přístup vychází z hardwarové implementace konečného automatu, kdy aktuální stav je uložen v registru a následující stav je vypočítáván pomocí kombinační logické sítě. Díky vhodnému kódování stavů je v architektuře vyřešeno odstranění nedeterminismu i realizace ε–přechodů. U architektury tak není nutné použít algoritmus 2.3 pro odstranění ε–přechodů a převod rozšířeného konečného automatu na nedeterministický automat, ale je možné mapovat do technologie FPGA přímo rozšířený konečný automat.

Při mapování automatu do FPGA je použito kódování one-hot. Pro každý stav je tak v architektuře vytvořen jednobitový flip-flop registr s tím, že stav je aktivní, pokud je v registru uložena logická jednička. Protože každý registr je možné nastavit nezávisle do logické jedničky nebo do logické nuly, je možné současně aktivovat více než jeden stav a procházet tak v automatu všechny nedeterministické cesty. Realizaci ε–přechodů je při tomto kódování stavů možné řešit pouhým propojením vstupů flip-flop registrů reprezentující stavy spojené ε–přechodem.

Přechody mezi stavy jsou v architektuře řešeny pomocí kombinační logické funkce. Ke každému přechodu je vytvořen komparátor, který porovnává vstupní znak a stav s definicí přechodu. Vytváří se tak signál indikující, jestli je přechod proveditelný, což umožňuje s využitím funkce *or* aktivovat následující stav. Celá přechodová tabulka je tak mapována na komparátory a hradla *OR*, přičemž každý přechod

je reprezentován právě jedním komparátorem. Příklad architektury je pro jednoduchý automat se třemi stavy znázorněn na obrázku 13.



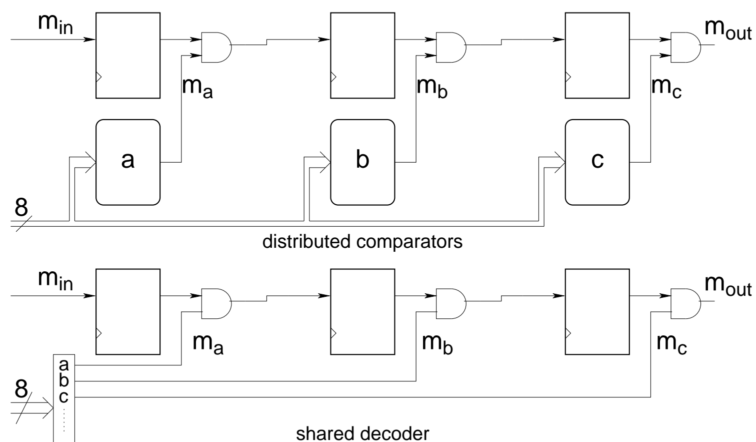
Obrázek 13: Příklad mapování nedeterministického automatu do FPGA.

V levé části obrázku je znázorněn automat a v pravé části jeho mapování na hardwarovou architekturu. Jednotlivým stavům *a*, *b*, *c* odpovídají stejně pojmenované jednobitové flip-flop registry. Jednička v registru znamená, že odpovídající stav je aktivní. Realizace jednotlivých přechodů je znázorněna stejným značením jako v automatu. Je patrná realizace jak ε–přechodů, tak i vícenásobných přechodů z jednoho stavu.

Uvedený přístup není možné použít na tvorbu NIDS s tisíci pravidly neboť pro každý přechod je použit jeden komparátor. Množství použitých hardwarových zdrojů tak s každým dalším řetězcem nebo regulárním výrazem velmi rychle narůstá. Další problém architektury je nízká frekvence, neboť do výpočtu následujícího stavu je nutné započítat i porovnání vstupního znaku.

3.12 Nedeterministický automat se sdíleným dekodérem

Mapování nedeterministického automatu do FPGA rozšířil Clark o sdílený dekodér znaků [17,18]. Sdílený dekodér redukuje počet použitých komparátorů podobným způsobem jako je tomu u architektury založené na zřetězených komparátorech. Znaky jsou v dekodéru převáděny na kód 1 z n a v podobě signálů distribuovány do kombinační logiky následujícího stavu. Jednotlivé přechody automatu tak nejsou řešeny komparátorem porovnávající vstupní znak, ale stačí porovnat pouze jednobitové signály z dekodéru. Rozdíl v mapování automatu s využitím komparátorů a s využitím sdíleného dekodéru je zachycen na obrázku 14.



Obrázek 14: Porovnání mapování nedeterministického automatu na architekturu s distribuovanými komparátory a na architekturu se sdíleným dekodérem.

Na obrázku je vidět, že u Clarkova přístupu je místo distribuovaných komparátorů použit jeden sdílený dekodér. Protože automaty pro hledání velkých množin regulárních výrazů obsahují mnohem více přechodů než je vstupních znaků, vede uvedená optimalizace na velkou redukci zdrojů na čipu a architekturu je tak možné použít pro mnohem větší množiny řetězců nebo regulárních výrazů než v případě distribuovaných komparátorů.

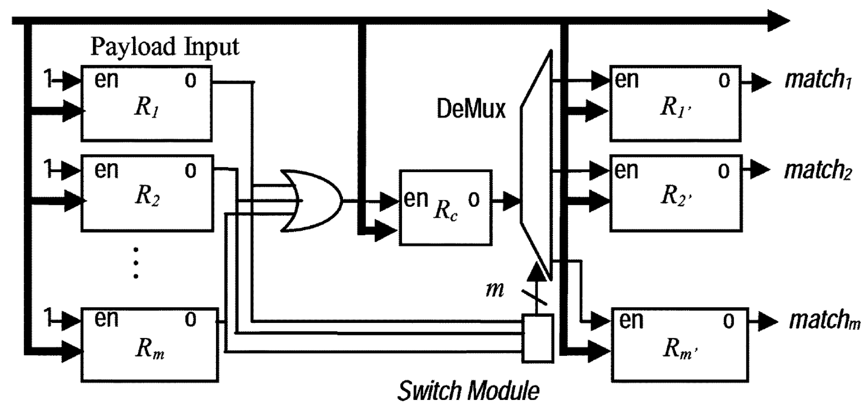
3.13 Sdílení logiky automatu pro prefixy, infixy a sufixy

Regulární výrazy pro popis signatur útoků často obsahují stejné prefixové, infixové nebo sufixové podvýrazy. Této skutečnosti využil Lin [34] k vytvoření optimalizace mapování nedeterministického automatu do technologie FPGA a rozšířil Clarkem navrženou architekturu o možnost sdílení infixových a sufixových výrazů.

Pokud mají dva regulární výrazy stejný prefix, je možné ve výsledném automatu sdílet stavy odpovídající shodnému prefixu. Automat pak přijímá stejný jazyk a současně je možné rozlišit, který regulární výraz byl nalezen, protože každému koncovému stavu odpovídá jeden regulární výraz. V případě sdílení prefixů je tak možné použít Clarkem navrženou hardwarovou architekturu.

V případě sdílení sufixů dochází ke spojování koncových stavů. Pokud automat dojde do koncového stavu, vzniká problém jak rozlišit, který regulární výraz byl nalezen. Při sdílení infixů mezi dvěma a více regulárními výrazy je situace ještě horší, neboť je potřeba zajistit, aby po přijetí infixového výrazu pokračovalo hledání v automatu, ze kterého se do infixového výrazu přešlo. Pokud by podmínka nebyla splněna a hledání by pokračovalo ve všech automatech sdílející daný infixový výraz, přijímal by výsledný automat jiný jazyk, než je definovaný zadanou množinou regulárních výrazů.

Autoři architektury v [34] řeší sdílení infixů tak, že si v registru zapamatují, který regulární výraz hledání infixového výrazu vyvolal. Na obrázku 15 je ukázána architektura pro sdílení infixového výrazu R_c pro celkem m regulárních výrazů $R_1, R_2, ...R_m$.



Obrázek 15: Sdílení infixů při mapování nedeterministického automatu do FPGA.

Pro mapování regulárních výrazů $R_1, R_2, ...R_m$ i R_c je využita hardwarová architektura navržena Clarkem. Pro všechny regulární výrazy je sdílen automat, který reprezentuje infixový výraz R_c. Ten je aktivován pomocí logického hradla *OR*, pokud byla u některého regulárního výrazu $R_1, R_2, ...R_m$ přijata část předcházející sdílenému infixu. Současně s aktivací infixového automatu je ve *Switch modulu* uloženo číslo regulárního výrazu, který infixový automat vyvolal. Pokud je infixový výraz přijat, aktivuje se podle uloženého čísla automat regulárního výrazu, který infixový automat vyvolal.

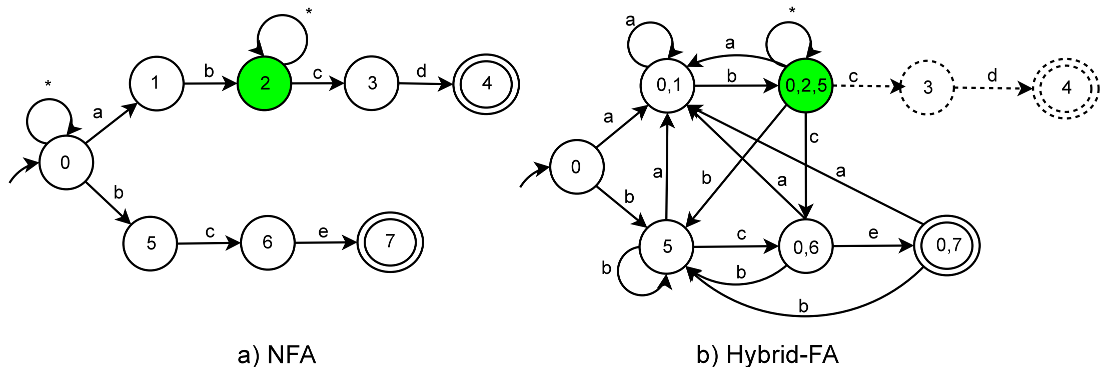
I když se v regulárních výrazech vyskytují shodné infixové výrazy, dosahuje architektura se sdílením infixů jen malé úspory zdrojů na čipu proti Clarkem navržené architektuře, pokud je před mapováním redukován automat s využitím sdílení prefixů.

3.14 Hybridní automaty

Jeden z důvodů velkého nárůstu paměťové složitosti deterministického automatu je konstrukce $.*$. Pokud některý regulární výraz obsahuje tuto konstrukci dochází k replikaci stavů, které odpovídají ostatním regulárním výrazům. S velikostí množiny regulárních výrazů narůstá i počet replikovaných stavů. Vliv $.*$ na velikost automatu je popsán v kapitole 3.8.

Hybridní automaty [8] se snaží redukovat paměťovou složitost způsobenou právě konstrukcemi $.*$. Jedná se o první přístup, který se snaží kombinovat výhody deterministického a nedeterministického automatu. K redukci paměťové složitosti dochází při determinizaci automatu. Pokud se při vytváření podmnožin stavů narazí na stav, který způsobuje stavovou explozi díky konstrukci $.*$, je vytváření přerušeno. Výsledkem je hybridní automat, který se skládá z jednoho deterministického a několika nedeterministických automatů.

Příklad vytvoření hybridního automatu pro regulární výrazy $ab.*cd$ a bce je zachycen na obrázku 16. Na obrázku je vidět, že vytváření podmnožin je přerušeno ve stavu 2, který je hraničním stavem mezi deterministickým a nedeterministickým automatem. Deterministický automat je označen plně, nedeterministický čárkovaně.



Obrázek 16: Nedeterministický a hybridní automat pro regulární výrazy $ab.*cd$ a bce. Deterministický automat je označen plně, nedeterministický čárkovaně.

Je vidět, že deterministický automat zahrnuje počáteční stav a jeho okolí. Nedeterministické automaty navazují na deterministický automat a jsou aktivovány v okamžiku, kdy je dosažen hraniční stav. Autoři se snaží tímto uspořádáním zajistit, aby většinu času pracoval pouze deterministický automat a nebylo nutné často přecházet do výrazně pomalejších nedeterministických automatů.

V hybridním automatu ale nelze vyloučit, že bude současně aktivních více nedeterministických automatů. Navíc v každém nedeterministickém automatu může být více současně aktivních stavů. Počet přístupů do tabulky přechodů by pak výrazně narostl a mohl by způsobit snížení propustnosti, neboť autoři předpokládají uložení tabulky přechodů v paměti. V nejhorším případě je časová složitost Hybridního automatu $O(N_{NFA} + 1)$, kde N_{NFA} je počet nedeterministických stavů a jednička odpovídá jednomu aktivnímu stavu deterministického automatu. Vzniká tak problém, jak garantovat propustnost vyhledávání, což může snadno využít útočník k zahlcení systému.

Nevýhodou hybridních automatů není jenom časová složitost, ale i omezení se pouze na redukci konstrukce $.*$. Zpětné smyčky a jiné nepříjemné vlastnosti regulárních výrazů nejsou hybridním automatem řešeny a mohou způsobovat enormní nárůst deterministického automatu nebo zvyšovat požadavky na propustnost paměti.

4 Analýza architektur pro hledání regulárních výrazů

Pro hledání regulárních výrazů se používají hardwarové architektury založené na deterministických a nedeterministických automatech. Výhodou deterministického automatu je zpracování vstupních dat v

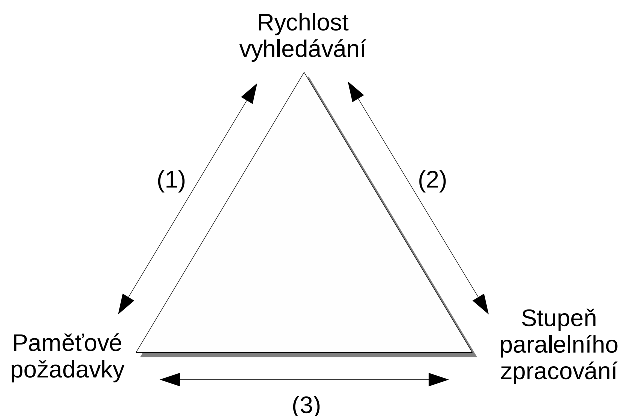
	regulární výraz délky n		m regulárních výrazů délky n	
	Časová složitost	Paměťová složitost	Časová složitost	Paměťová složitost
NFA	$O(n^2)$	$O(n)$	$O((nm)^2)$	$O(nm)$
DFA	$O(1)$	$O(\Sigma^n)$	$O(1)$	$O(\Sigma^{nm})$

Tabulka 2: Časová složitost přijetí jednoho vstupního symbolu a paměťová složitost pro nedeterministický a deterministický konečný automat. U nedeterministického automatu se počítá s odstraněním nedeterminismu pomocí backtrackingu.

lineárním čase. V každém hodinovém cyklu je zpracován jeden vstupní symbol, což znamená konstantní rychlost vyhledávání a tedy i garantovanou propustnost. Nevýhodou deterministického automatu je velká paměťová náročnost. Vlivem determinizace může dojít teoreticky až k exponenciálnímu nárůstu počtu stavů automatu, což způsobuje exponenciální nárůst velikosti přechodové tabulky. Vzniká tak problém najít dostatečně rychlou a dostatečně velkou paměť pro uložení tabulky přechodů. Nedeterministické automaty mají sice lineární paměťovou složitost s ohledem na délku regulárního výrazu, ale při odstranění nedeterminismu pomocí zpětného vyhledávání (backtrakingu) je pro přijetí jednoho vstupního symbolu potřeba kvadratická časová složitost $O(n^2)$, kde n je počet stavů automatu. Časovou a paměťovou složitost deterministického a nedeterministického automatu uvádí přehledně tabulka 2. V případě nedeterministického automatu se předpokládá odstranění nedeterminismu s využitím backtrackingu.

Současné architektury síťových ale i konvenčních procesorů nabízejí možnost paralelního zpracování. U konvenčních procesorů najdeme jednotky nebo desítky jader, v síťových procesorech jsou použity řádově desítky procesních jednotek (microengines) [23]. Paralelního zpracování je možné využít i u čistě hardwarových technologií. Například technologie FPGA obsahuje řádově statisíce registrů a look-up tabulek, které mohou pracovat paralelně.

S využitím paralelního zpracování je možné urychlit vyhledávání regulárních výrazů nebo snížit paměťové požadavky. Triviální způsob urychlení je distribuce vstupních dat mezi více paralelních jednotek, což sebou přináší nemalou režii spojenou s distribucí dat mezi jednotky a replikaci datových struktur. Řada přístupů se proto snaží využít paralelní zpracování přímo na úrovni automatu. Například u nedeterministického automatu je možné procházet paralelně k nedeterministických cest, omezit backtracking a snížit tak časovou složitost na $O((\frac{nm}{k})^2)$. U deterministického automatu je možné rozdělit množinu regulárních výrazů na k podmnožin a hledat každou podmnožinu pomocí jiné jednotky, což redukuje paměťovou složitost na $O(k \cdot \Sigma^{\frac{m}{k}n})$. Paralelní zpracování se dá také využít pro kompresi přechodové tabulky nebo pro překódování vstupních symbolů.



Obrázek 17: Vzájemná závislost mezi rychlostí vyhledávání, velikostí paměti a stupněm paralelního zpracování.

Rychlost vyhledávání, paměť a stupeň paralelního zpracování spolu úzce souvisejí. Kompromis mezi těmito parametry charakterizuje obrázek 17. Na obrázku jsou znázorněny celkem tři závislostí:

1. Závislost rychlosti vyhledávání a velikosti paměti při konstantním stupni paralelního zpracování. Determinizací automatu je možné dosáhnout vyšší rychlost vyhledávání, ale jen za cenu zvýšení požadavků na kapacitu paměti. Naopak pokud je v některé části nebo v celém automatu zachován nedeterminismus, je potřeba k uložení automatu méně paměti než v případě deterministického automatu, ale současně může docházet k backtrackingu, což znamená snížení rychlosti vyhledávání.

2. Závislost rychlosti vyhledávání na stupni paralelního zpracování při stejné velikosti paměti (stejná velikost datové struktury). Díky paralelnímu zpracování je možné procházet v grafu přechodů automatu více nedeterministicky určených cest, což omezuje backtracking a je možné dosáhnout vyšší rychlosti vyhledávání. Naopak při nižším stupni paralelního zpracování se více projevuje backtracking a dochází ke snížení rychlosti vyhledávání.

3. Závislost paměti na stupni paralelního zpracování při zachování konstantní rychlosti vyhledávání. Při stejné rychlosti vyhledávání je možné vyšším stupněm paralelního zpracování dosáhnout zmenšení velikosti datové struktury například povolením nedeterminismu nebo zavedením komprese přechodové tabulky. Naopak pokud se sníží stupeň paralelního zpracování, je možné dosáhnou stejné rychlosti vyhledávání jen za cenu nárůstu paměťových požadavků. Například je možné provést determinizaci části nebo celého automatu.

Ze vzájemných závislostí je zřejmé, že výběr vhodného algoritmu nebo vhodné architektury silně závisí na vlastnostech cílové technologie a požadavcích kladených na propustnost. Je důležité se soustředit zejména na vlastnosti cílové technologie, jako je stupeň paralelního zpracování nebo rychlost a kapacita dostupných pamětí. U konvenčních procesorů jsou k dispozici jednotky nebo desítky jader a vyrovnávací paměť cache dosahuje velikosti až 16 MB. V případě technologie FPGA je možné využít masivně paralelní zpracování, ale jen s omezenou pamětí, která u této technologie v současné době dosahuje velikosti kolem jednoho megabajtu.

Díky malému stupni paralelního zpracování na konvenčních ale i síťových procesorech, vznikla pro problematiku hledání regulárních výrazů řada přístupů, které jsou založeny na deterministických automatech. Deterministické automaty dosahují vysokých rychlostí vyhledávání jen za cenu velké paměťové náročnosti, proto se jednotlivé přístupy snaží redukovat jejich paměťovou složitost. Fang Yu ukázala, že rozdělením množiny regulárních výrazů na podmnožiny a konstrukcí několika paralelních deterministických automatů je možné redukovat nárůst počtu stavů. Další významnou redukci velikosti přechodové tabulky deterministického automatu představil Sallesh Kumar, který s využitím implicitních přechodů, modelu Delay DFA a H-FA výrazně snížil počet přechodů deterministického automatu. Aby nedošlo ke zpomalení vyhledávání, vyžaduje Kumarem navržený přístup více paralelních jednotek a distribuci přechodové tabulky mezi více pamětí. Významné redukce paměti dosáhla i Becchi s využitím hybridního automatu, ale u navrženého přístupu je konstruována řada nedeterministických automatů a není tak možné garantovat propustnost.

I když navržené redukce snižují paměťovou složitost, stále jsou kladeny vysoké požadavky na kapacitu a propustnost paměti pro uložení tabulky přechodů. Datová struktura se i přes redukce často nevejde do vyrovnávací paměti cache, což snižuje rychlost vyhledávání.

Přístupy založené na deterministických automatech nejsou moc vhodné pro mapování do technologie FPGA, neboť není využit masivní paralelismus, který technologie nabízí. Navíc navržené redukce vyžadují několik paralelních pamětí nebo víceportovou paměť. Pro uložení datových struktur je tak možné použít rychlou paměť na čipu nebo několik pamětí mimo čip. U paměti na čipu vzniká problém s omezenou kapacitou, kdy i po redukci přesahuje velikost přechodové tabulky kapacitu všech dostupných pamětí na čipu. U externích pamětí je zase potřeba uvažovat s latencí, která prodlužuje dobu výpočtu nového stavu a tím snižuje rychlost vyhledávání.

Pro technologii FPGA proto vznikla řada přístupů, které se snaží mapovat na hardwarovou architekturu přímo nedeterministický automat. U nedeterministického automatu narůstá počet stavů lineárně s délkou regulárního výrazu. K uložení celé tabulky přechodů je potřeba mnohem méně paměti, než v případě deterministického automatu.

U nedeterministického automatu ale nejde na základě aktuálního stavu a vstupního symbolu vždy jednoznačně určit následující stav. Nedeterministický výběr následujícího stavu je možné řešit pomocí backtrackingu nebo Thomsonovou simulací. Při backtrackingu se na základě stavu a vstupního symbolu vybere jeden z možných následujících stavů a pokud neexistuje cesta do koncového stavu, vrací se automat ve vstupních datech zpět. Thomsonova simulace umožňuje aktivovat více stavů současně. V každém kroku se z množiny aktuálních stavů a vstupního symbolu počítá nová množina následujících stavů. Procházejí se tak všechny nedeterministické cesty současně. Někdy se Thomsonově simulaci také říká determinizace automatu za běhu (on-line determinizace).

Při backtrackingu dochází ke snížení rychlosti vyhledávání a není možné garantovat určitou propustnost. Při Thomsonově simulaci je možné plynule zpracovávat vstupní data a garantovat konstantní rychlost vyhledávání jen za předpokladu, že pro všechny aktivní stavy je zajištěn výpočet množiny následujících stavů v konstantním čase (jednom hodinovém cyklu). Z tabulky přechodů je ale v takovém případě potřeba přečíst stejný počet přechodů, jako je aktivních stavů. Z toho vyplývá, že tabulka přechodů musí být uložena v paměti s dostatečnou propustností, kterou je pro automat pracující na frekvenci f, s maximálním počtem aktivních stavů $|Q_{max}|$ a velikostí paměti $|\delta|$ pro uložení jednoho přechodu přechodové funkce δ možné vyjádřit jako

$$T_{mem} = |\delta| \cdot |Q_{max}| \cdot f \tag{2}$$

Z rovnice 2 je vidět, že požadovaná propustnost paměti závisí na maximálním počtu současně aktivních stavů $|Q_{max}|$, což je hodnota, která se může pro různé automaty výrazně lišit. Další problém je, že v případě aktivace velkého počtu stavů může dojít k extrémním požadavkům na propustnost paměti. Pro realizaci nedeterministických automatů je z těchto důvodů vhodná technologie FPGA, neboť umožňuje pro konkrétní automat přizpůsobit architekturu tak, aby rychlost paměti nebyla úzkým místem. Navíc FPGA poskytuje možnost masivního paralelismu, takže je možné zajistit výpočet následujících stavů i při aktivaci velkého počtu stavů.

Aby přístupy k tabulce přechodů nebyly úzkým místem při mapování nedeterministického automatu do FPGA, využívají současné architektury pro reprezentaci tabulky přechodů místo paměti přímo logiku na čipu. Základní princip mapování nedeterministického automatu do technologie FPGA představili Sidhu a Prasana. V autory navržené architektuře je pro každý stav vytvořen flip-flop registr s tím, že stav je aktivní, pokud je v registru uložena logická jednička. Protože každý registr je možné nastavit nezávisle do logické jedničky nebo do logické nuly, je možné mít současně aktivní prakticky libovolnou podmnožinu stavů a řešit efektivně nedeterminismus aktivací všech nejednoznačně určených následujících stavů. Přechody mezi stavy jsou v architektuře řešeny pomocí kombinační logické funkce, která je mapována do look-up tabulek. Ke každému přechodu je vytvořen komparátor, který porovnává vstupní znak a stav s definicí přechodu. Vytváří se tak signál indikující, jestli je přechod proveditelný, což umožňuje s využitím funkce *OR* aktivovat následující stav. Celá přechodová tabulka je tak mapována na komparátory a hradla *OR*, přičemž každý přechod je reprezentován právě jedním komparátorem.

Nevýhodou uvedeného mapování je využití velkého množství flip-flop registrů a look-up tabulek, které jsou drahé v porovnání s obyčejnou pamětí. Proto se Clark snažil snížit množství spotřebovaných zdrojů zavedením sdíleného dekodéru, který překódovává vstupní znaky na jednobitové signály. Přechody automatu pak nemusí být řešeny komparátorem na celý vstupní znak, ale stačí porovnávat pouze jednobitové signály z dekodéru. Protože automaty pro hledání velkých množin regulárních výrazů obsahují mnohem více přechodů než je vstupních znaků, vede uvedená optimalizace na velkou redukci zdrojů na čipu. Další vylepšení mapování nedeterministického automatu na FPGA se snaží využít posuvných registrů pro reprezentaci podřetězců a šetřit zdroje, pokud více regulárních výrazů obsahuje stejný podvýraz. Obě optimalizace ale přinášejí u velkých množin regulárních výrazů jen minimální redukci zdrojů na čipu.

Mapování nedeterministického automatu umožňuje procházet v automatu paralelně všechny nedeterministické cesty. Ke každému stavu je generován jeden flip-flop registr, což umožňuje řešit nedeterminismus mezi libovolnou podmnožinou stavů automatu, dokonce i mezi množinu všech stavů. I když u nedeterministického automatu může docházet k aktivaci více stavů najednou, je velká část stavů vždy neaktivní. Flip-flop registry neaktivních stavů a jim odpovídající logika následujícího stavu není v daný okamžik využita pro výpočet následujícího stavu.

Aby bylo možné zjistit, jak efektivně je využita logika na čipu, je potřeba analyzovat, kolik stavů může být v nedeterministickém automatu současně aktivních. Takovou informaci je možné získat ze vzájemné

	L7 dek.	Snort 1	Snort 2	Snort 3	Snort 4	Snort 5
Celkem stavů NFA [-]	774	3888	2774	1060	1038	819
Max. aktivních stavů [-]	23	122	19	18	25	32
Max. aktivních stavů [%]	2,97	3,14	0,68	1,70	2,41	3,91

Tabulka 3: Počet maximálně aktivních stavů v nedeterministických automatech, které reprezentují regulární výrazy programu L7 dekodér a pět vybraných modulů programu Snort.

relace mezi modely nedeterministického a deterministického automatu, které byly popsány v kapitole 2. V deterministickém automatu $A^D = (N^D, \Sigma^D, \delta^D, q_0^D, F^D)$ je každý stav $q^D \in Q^D$ definován množinou stavů nedeterministického automatu $A^N = (N, \Sigma, \delta, q_0, F)$. Z konstrukce deterministického automatu pomocí algoritmu 2.4 plyne, že pro každou množinu stavů, která může být aktivována v nedeterministickém automatu, existuje odpovídající stav v deterministickém automatu. To znamená, že můžeme najít stav q^D_{max}, který odpovídá maximálnímu počtu aktivních stavů v nedeterministickém automatu, a pro který platí:

$$\forall q^D \in Q^D : |q^D_{max}| \geq |q^D| \tag{3}$$

Pro regulární výrazy programu L7 dekodér a pro vybraných pět modulů volně dostupného programu Snort byla provedena analýza, kolik může být v automatu vytvořeném z regulárních výrazů maximálně současně aktivních stavů. Množiny regulárních výrazů byly převedeny algoritmem 2.1 na ekvivalentní nedeterministický automat. Následně byla provedena determinizace pomocí algoritmu 2.4 a z výsledného automatu byl podle rovnice 3 vybrán stav, který reprezentuje maximální počet aktivních stavů v nedeterministickém automatu. Výsledky analýzy jsou pro všechny množiny regulárních výrazů shrnuty v tabulce 3.

V tabulce je pro množiny regulárních výrazů v prvním řádků uveden celkový počet stavů nedeterministického automatu, v druhém maximální počet aktivních stavů a ve třetím kolik procent z celkového počtu stavů může být maximálně aktivních. Z výsledků je vidět, že pro všechny analyzované množiny regulárních výrazů je současně aktivních méně než 4 % stavů. To znamená, že pro výpočet následujícího stavu se při zpracování jednoho vstupního symbolu použije méně než 4 % zdrojů FPGA, zbylých 96 % zůstává nevyužito.

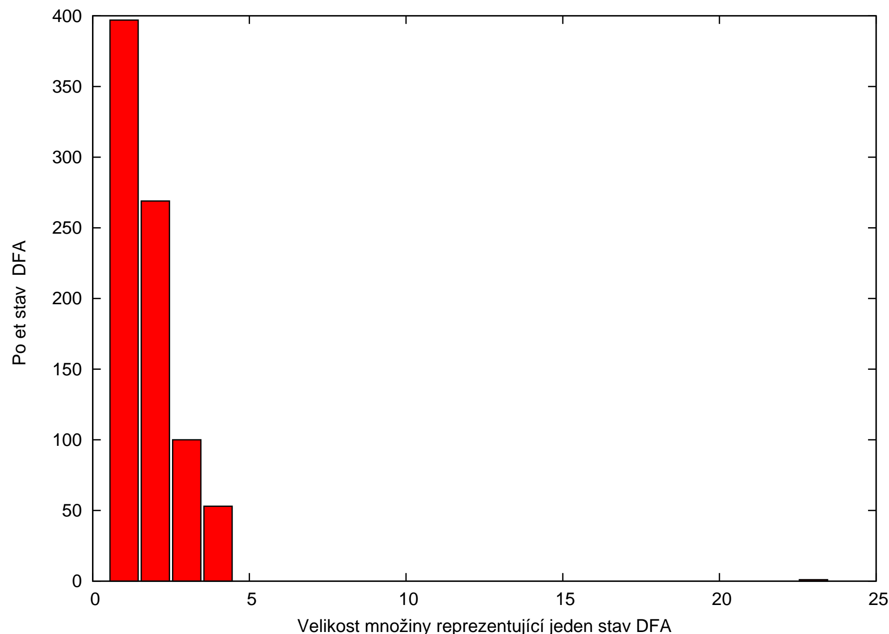
Protože pro každou množinu stavů, která může být aktivována v nedeterministickém automatu, existuje odpovídající stav v deterministickém automatu, je možné analyzovat jak velké množiny stavů mohou být v nedeterministickém automatu současně aktivní a kolik takových množin existuje. Pro množinu regulárních výrazů programu L7 dekodér byl proto vytvořen histogram stavů deterministického automatu, který je vidět na obrázku 18. V histogramu jsou všechny stavy automatu $q^D \in Q^D$ rozděleny do sloupců podle velikosti množiny $|q^D|$, která daný stav definuje množinou současně aktivních stavů v nedeterministickém automatu.

Z histogramu na obrázku 18 je vidět, že většina stavů deterministického automatu je definována množinou pouze jednoho nebo dvou stavů. U současných architektur mapujících nedeterministický automat do technologie FPGA je tak ve většině případů aktivní jeden nebo dva flip-flop registry a na čipu se využívá pouze jim odpovídající kombinační logika následujícího stavu. Pro všechny analyzované množiny regulárních výrazů navíc mohou současně aktivní stavy tvořit jen velmi malou část automatu, což dává prostor pro nové efektivní mapování nedeterministických automatů do technologie FPGA.

Reference

[1] Alfred V. Aho and Margaret J. Corasick. Efficient String Matching: An Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Michael Attig and John Lockwood. Sift: Snort intrusion filter for tcp. *High-Performance Interconnects, Symposium on*, 0:121–127, 2005.



Obrázek 18: Histogram stavů deterministického automatu podle počtu prvků množiny, která daný stav definuje.

[3] Z. K. Baker and V. K. Prasanna. Automatic Synthesis of Efficient Intrusion Detection Systems on FPGAs. In *Proceedings of the 14th Annual International Conference on Field-Programmable Logic and Applications (FPL '04)*, 2004.

[4] Zachary K. Baker and Viktor K. Prasanna. A methodology for synthesis of efficient intrusion detection systems on fpgas. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 135–144, Washington, DC, USA, 2004. IEEE Computer Society.

[5] Zachary K. Baker and Viktor K. Prasanna. Time and Area Efficient Pattern Matching on FPGAs. In *FPGA '04: Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays*, pages 223–232, New York, NY, USA, 2004. ACM Press.

[6] Zachary K. Baker and Viktor K. Prasanna. High-throughput linked-pattern matching for intrusion detection systems. In *ANCS '05: Proceedings of the 2005 ACM symposium on Architecture for networking and communications systems*, pages 193–202, New York, NY, USA, 2005. ACM.

[7] Zachary K. Baker and Viktor K. Prasanna. Automatic synthesis of efficient intrusion detection systems on fpgas. *IEEE Trans. Dependable Secur. Comput.*, 3(4):289–300, 2006.

[8] Michela Becchi and Patrick Crowley. A hybrid finite automaton for practical deep packet inspection. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, New York, NY, December 2007. ACM.

[9] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154, New York, NY, USA, 2007. ACM.

[10] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: Theory to practice. In *Proceedings of the 2008 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*. ACM, December 2008.

[11] Michela Becchi and Patrick Crowley. Extending finite automata to efficiently match perl-compatible regular expressions. In *Proceedings of the International Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Madrid, Spain, December 2008. ACM.

[12] B. H. Bloom. Space/Time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the Association for Computing Machinery*, 13(7):422–6, July 1970.

[13] R. S. Boyer and J. S. Moore. A Fast String Searching Algorithm. *Communications of the Association for Computing Machinery*, 20(10):762–772, October 1977.

[14] C. Charras and T. Lecroq. Exact string matching algorithms. Technical report, Université de Rouen, leden 1997.

[15] Young H. Cho and William H. Mangione-Smith. Deep Packet Filter with Dedicated Logic and Read Only Memories. In *12th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pages 125–134, Napa, CA, 2004.

[16] Inc. Cisco Systems. Cisco IOS Intrusion Prevention Systems. Dokument dostupný na http://www.cisco.com, 2005.

[17] Ch. Clark and D. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *Field Programmable Logic and Application, 13th International Conference*, pages 956–959, Lisbon, Portugal, 2003.

[18] Christopher R. Clark and David E. Schimmel. Scalable Pattern Matching for High-Speed Networks. In *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 249–257, Napa, California, 2004.

[19] S. Dharmapurikar, P. Krishnamurthy, T. S. Sproull, and J. W. Lockwood. Deep Packet Inspection using Parallel Bloom Filters. *IEEE Micro*, 24(1):52–61, 2004.

[20] Sarang Dharmapurikar, Praveen Krishnamurthy, Todd Sproull, and John Lockwood. Deep packet inspection using parallel bloom filters. *High-Performance Interconnects, Symposium on*, 0:44, 2003.

[21] Sarang Dharmapurikar and John W. Lockwood. Fast and scalable pattern matching for network intrusion detection systems. *IEEE Journal on Selected Areas in Communications*, 24(10):1781–1792, 2006.

[22] Domenico Ficara, Stefano Giordano, Gregorio Procissi, Fabio Vitucci, Gianni Antichi, and Andrea Di Pietro. An improved dfa for fast regular expression matching. *SIGCOMM Comput. Commun. Rev.*, 38(5):29–40, 2008.

[23] Mark A. Franklin, Patrick Crowley, Haldun Hadimioglu, and Peter Z. Onufryk. *Network Processor Design, Volume 3: Issues and Practices, Volume 3*. Morgan Kaufmann, 2005.

[24] John E. Hopcroft and Jefferey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Adison-Wesley Publishing Company, Reading, Massachusets, USA, 1979.

[25] Inc. Juniper Networks. Intrusion Detection and Prevention. Dokument dostupný na http://www.juniper.net/solutions/literature/white_papers/wp_idp.pdf, 2005.

[26] Inc. Juniper Networks. NetScreen-IDP 10/100/500/1000 Specifications. Dokument dostupný na http://www.juniper.net/products/intrusion/dsheet/110010.pdf, 2005.

[27] Inc. Juniper Networks. Summary of Attacks & Attack Detection Methods. Dokument dostupný na http://www.juniper.net, 2005.

[28] R. M. Karp and M. O. Rabin. Efficient Randomized Pattern-Matching Algorithms. Technical Report TR-31-81, Harvard University, MA, USA, 1981.

[29] D. E. Knuth, J. H. Morris Jr., and V. R. Pratt. Fast Pattern Matching in Strings. *SIAM J. Comput*, 6(2):323–350, 1977.

[30] Sailesh Kumar, Balakrishnan Chandrasekaran, Jonathan Turner, and George Varghese. Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In *ANCS '07: Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 155–164, New York, NY, USA, 2007. ACM.

[31] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 339–350, New York, NY, USA, 2006. ACM.

[32] Sailesh Kumar, Jonathan Turner, and John Williams. Advanced algorithms for fast and scalable deep packet inspection. In *ANCS '06: Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems*, pages 81–92, New York, NY, USA, 2006. ACM.

[33] Mark V. Lawson. *Finite Automata*. Chapman and Hall, London, 2003. 320 s.

[34] Cheng-Hung Lin, Chih-Tsun Huang, Chang-Ping Jiang, and Shih-Chieh Chang. Optimization of pattern matching circuits for regular expression on fpga. *IEEE Trans. Very Large Scale Integr. Syst.*, 15(12):1303–1310, 2007.

[35] Tomáš Martínek, Jan Kořenek, and Jiří Novotný. Network monitoring adaptor for 10gbps technology using fpga. In *CESNET Conference 2006 Proceedings*, pages 143–151. CESNET National Research and Education Network, 2006.

[36] Inc. McAfee. McAfee IntruShield Network IPS Appliances datasheet. Dokument dostupný na http://www.mcafeesecurity.com, 2005.

[37] Inc. Micron Technology. Harmony 2M and 1M Ternary CAMs, březen 2002.

[38] V. Paxson, K. Asanović, S. Dharmapurikar, J. Lockwood, R. Pang, R. Sommer, and N. Weaver. Rethinking hardware support for network analysis and intrusion prevention. In *HOTSEC'06: Proceedings of the 1st USENIX Workshop on Hot Topics in Security*, pages 11–11, Berkeley, CA, USA, 2006. USENIX Association.

[39] SCAMPI. SCAMPI (IST-2001-32404) Project WWW Page. http://www.ist-scampi.org, 2005.

[40] David V. Schuehler, James Moscola, and John W. Lockwood. Architecture for a hardware-based, tcp/ip content-processing system. *IEEE Micro*, 24(1):62–69, 2004.

[41] R. Sidhu and V. K. Prasanna. Fast Regular Expression Matching using FPGAs. In *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2001)*, pages 227–238, April 2001.

[42] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast hash table lookup using extended bloom filter: an aid to network processing. In *SIGCOMM '05: Proceedings of the 2005 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 181–192, New York, NY, USA, 2005. ACM.

[43] I. Sourdis and D. N. Pnevmatikatos. Fast, Large-Scale String Match for a 10Gbps FPGA-Based Network Intrusion Detection System. In *Field Programmable Logic and Application, 13th International Conference*, pages 880–889, Lisbon, Portugal, 2003.

[44] I. Sourdis, D.N. Pnevmatikatos, and S. Vassiliadis. Scalable multigigabit pattern matching for packet inspection. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 16, Issue 2*, pages 156–166, February 2008.

[45] Ioannis Sourdis and Dionisios Pnevmatikatos. Pre-decoded cams for efficient and high-speed nids pattern matching. In *FCCM '04: Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 258–267, Washington, DC, USA, 2004. IEEE Computer Society.

[46] Lin Tan, Brett Brotherton, and Timothy Sherwood. Bit-split string-matching engines for intrusion detection and prevention. *ACM Trans. Archit. Code Optim.*, 3(1):3–34, 2006.

[47] Lin Tan and Timothy Sherwood. Architectures for bit-split string scanning in intrusion detection. *IEEE Micro*, 26(1):110–117, 2006.

[48] Ken Thompson. Programming techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[49] Xilinx. DS031 Virtex-II 1.5V Field-Programable-Gate-Arrays, září 2002. Dokument dostupný na http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[50] Milan Češka. Teoretická informatika, září 2002.