软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software,2011,22(8):1838−1854 [doi: 10.3724/SP.J.1001.2011.04034]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel/Fax: +86-10-62562563

面向网络安全的正则表达式匹配技术*

张树壮[1+], 罗 浩[2], 方滨兴[1,2]

[1](哈尔滨工业大学 计算机科学与技术学院,黑龙江 哈尔滨 150001)

[2](中国科学院 计算技术研究所 信息安全研究中心,北京 100190)

Regular Expressions Matching for Network Security

ZHANG Shu-Zhuang[1+], LUO Hao[2], FANG Bin-Xing[1,2]

[1](School of Computer Science and Technology, Harbin Institue of Technology, Harbin 150001, China)

[2](Information Security Research Center, Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100190, China)

+ Corresponding author: E-mail: zhangshuzhuang@pact518.hit.edu.cn, http://pact518.hit.edu.cn

Zhang SZ, Luo H, Fang BX. Regular expressions matching for network security. *Journal of Software*, 2011, 22(8):1838−1854. http://www.jos.org.cn/1000-9825/4034.htm

Abstract: This paper analyzes the regular expression matching methods' time complexity, space complexity and the tradeoff between them. The experiences, problems, and challenges encountered by the regular expression matching in network security field are well-classified and discussed in depth. Focusing on the two issues, a comprehensive overview of the current optimizing techniques and strategies adopted by academic and business communities is presented. Finally, a conclusion and some suggestions for future research are put forward.

Key words: signature matching; deep packet inspection; regular expression; finite automata; memory reduction

摘 要: 分析了基于有穷状态自动机的正则表达式匹配方法的时间复杂度、空间复杂度以及二者之间的制约关系,深入讨论了在网络安全应用中遇到的特有问题与挑战.围绕这两个问题,对当前出现的多种优化技术和策略进行了全面的综述和评价,最后对未来的研究方向进行了总结和展望.

关键词: 特征匹配;深度包检测;正则表达式;有穷自动机;内存缩减

中图法分类号: TP393 **文献标识码**: A

传统的网络安全检测是对数据包的结构化头部进行分析.然而随着网络的不断发展,许多病毒、恶意代码、入侵指令、垃圾邮件等信息都隐藏在数据包的内容之中.因此,当前在进行安全检测时,除了要对数据包头部进行检查之外,也要对数据包的内容进行检测.例如,入侵检测系统会针对各种入侵行为的特点建立一组入侵行为的特征模式集,并定义针对各种行为所应该采取的措施.这些行为特征和对应的措施构成系统的安全规则,当发现给定数据命中系统的某条特征模式时则采取相应的措施.病毒检测系统有一个庞大的病毒特征库.据统计,到2010 年 6 月,网络病毒已接近 758 473 万种.病毒检测的主要操作就是在每一个文件中搜索病毒库中的特征.

* 基金项目: 国家自然科学基金(60903209); 国家重点基础研究发展计划(973)(2007CB311100); 国家高技术研究发展计划(863)(2009AA01Z437, 2007AA01Z406, 2007AA01Z467, 2007AA01Z442, 2007AA01Z474, 2011AA012504)

收稿时间: 2010-03-22; 修改时间: 2010-10-11; 定稿时间: 2011-03-07

CNKI 网络优先出版: 2011-05-12 11:47，http://www.cnki.net/kcms/detail/11.2560.TP.20110512.1147.005.html

这种使用一组给定的“特征”与网络中的数据内容进行比较,从而发现其中的恶意特征的方法称为深度包检测(deep packet inspection).对于网络安全应用来说,“特征”通常是指入侵检测、防病毒、垃圾邮件过滤等应用中定义的模式串,用来表示攻击、病毒或者垃圾邮件区别于正常网络流量的特征.最初的安全规则以精确字符串为主.然而,随着网络的不断发展,网络安全攻防双方的技术也在不断发展.为了躲避检测,各种攻击、恶意代码等又都在不断刻意隐藏自己的特征,这使得安全系统中所需要的检测规则也越来越复杂.例如,snort[1]规则:Alert tcp $HOME_NET any→$EXTERNAL_NET 1863(msg:“CHAT MSN login attempt”;flow:to_server,established; content:“USR”;depth:4;nocase;content:“TWN”;distance:1;nocase;)表示要同时匹配“USR”和“TWN”这两个关键词;同样,对于原本为简单字符串的关键词,在实际中也可能以各种不同的形式出现(例如,“模式匹配”可能会写成“模#式#匹#配”).这就需要逐一识别各个部分后再进行一定限定条件(位置顺序、有限间距等)的判断来进行识别.可见,为了表达更精确的语义和上下文信息,需要将多个特征按照一定的顺序和限定条件进行组合,形成一条复合的匹配规则.上述这些问题所需要的规则形式无法用精确串来描述,因此,经典的多模串匹配算法如AC[2],SBOM[3],WU-MANBER[4]等也无法直接使用.

当人们发现应用层上各种复杂的协议和网络中的攻击方式已经越来越难以提取出准确的字符串特征时,正则表达式以其强大、灵活的表达能力,迅速成为描述新一代规则的主要工具.商业界和开源社区都开始广泛使用正则表达式来增强协议特征和安全特征的描述.例如,Linux 应用层协议分类器(linux application protocol classifier,简称 L7-filter)[5]就全部采用正则表达式来识别 100 多种应用层协议,其中包含了多个复杂的 P2P 协议.开源的入侵检测系统 Snort 在 2003 年以前,检测规则全部为精确字符串特征,但最新发布的规则集中,正则表达式的比例已经超过了 40%.Bro 入侵检测系统[6]也直接使用正则表达式来描述它的规则集.在商业市场上,3Com 的 TippingPoint X505[7]以及 Cisco 的各种网络安全系统[8]都开始使用正则表达式.目前,Cisco 已经将基于正则表达式的内容检测集成到了其网络操作系统中.在研究领域,UC Berkeley,bell-labs 以及 google 的 fang yu 等人[9]论述了正则表达式在网络安全检测和协议识别中的优势,对不同的匹配方法进行了分析和评价,并提出了规则改写和分组匹配等方法.随后,由于 cisco 公司的推动,Washington University 的 Sailesh Kumar,Michela Becchi 等人对这个问题进行了更为深入和细致的研究,提出了 D^2FA[10]、状态合并(state merging)[11]、H-FA[12]、混合自动机(hybrid FA)[13]以及 XFA[14]等方法来实现大规模正则表达式的实用化.这些工作大大提高了某些特殊类型的正则表达式的实用化和匹配效率.国内学术界对高性能正则表达式匹配技术的研究也在不断加强,哈尔滨工业大学网络与信息安全技术研究中心、清华大学、国防科学技术大学、中国科学院计算技术研究所信息安全中心等,都在逐渐投入到这一关键技术的研究中.

使用正则表达式来描述应用层各种协议和攻击的特征,比传统的提取精确匹配字符串方法更准确、更方便、更有效,然而,其强大的能力也使得它在实际应用中面临诸多的挑战.本文第 1 节给出正则表达式匹配的定义并对各种匹配方法进行分析和比较,最后着重强调正则表达式在网络安全应用中所面临的挑战.第 2 节~第 4 节详细介绍正则表达式匹配技术亟待深入研究的两个关键问题:空间缩减和性能保证,详细论述解决这些问题的多种方法及策略,对其优缺点进行评价.第 5 节对全文进行总结,并指出未来研究的若干方向.

1 问题描述

1.1 正则表达式的匹配方法

有穷自动机(finite automaton,简称 FA)和正则表达式所表示的语言都是正则语言[15],因此通常用来实现对正则表达式的匹配.有两种典型的有穷自动机可以完成这个任务:非确定型有穷自动机(nondeterministic finite automaton,简称 NFA)和确定型有穷自动机(deterministic finite automaton,简称 DFA).一个 DFA 包括 5 个部分:(1) 有穷状态集合 Q;(2) 有穷的输入符号集合Σ,又称为字母表;(3) 转换函数δ,又称为跳转函数,它以一个状态和一个输入符号作为变量,返回值为一个状态;(4) 初始状态 $s_0(s_0\in Q)$;(5) 接受状态集合 F.一个 NFA 同样包含这 5 个部分.NFA 与 DFA 的区别在于,DFA 中δ的返回值确定为 1 个状态;NFA 中δ是一个以状态和输入字符为变量的函数,但是返回值是可能包含 0 个或多个状态的集合.NFA 和 DFA 之间可以进行相互转化,但是在从 NFA

到 DFA 的转化过程中,可能会出现状态数目的剧烈增长,称为状态“爆炸”.在本节的后面将会详细说明这种情况.

实际应用中,由于对需要处理的数据没有任何先验知识,因此无法确定是否存在一个子串匹配给定规则或者匹配的子串从何处开始.在这种情况下,有两种方法来正确实现搜索:重复搜索和“一遍搜索(one pass search)”.重复搜索的具体做法为:用给定表达式直接构建自动机,然后以输入字符串的每个字符为开始位置重复搜索.在每次搜索过程中,顺次读取并处理字符,直到找出了所有的匹配子串.下一次搜索从上一次起始点的下一个字符开始(如果匹配类型为穷举匹配)或从最后一个匹配子串的下一个字符开始(如果匹配类型为非重叠匹配).重复搜索通常用于语言的解析中.因为可能要对同一个字符串进行多次搜索,所以其效率非常低.在安全检测类应用中,通常使用“一遍搜索”法.其做法为,将“.*”放在没有“^”标记的表达式的开头,然后再构建成自动机.这就意味着表达式所表示的模式可以出现在输入数据的任何地方.只要从前到后一次扫描并处理每一个字符,就可以识别出从任何位置开始的匹配子串.本文后面的讨论都是针对这种情况.

1.2 正则表达式匹配方法所面临的挑战

在实际的应用系统中,通常都配置了多条规则.例如,l7-filter 的正则表达式规则为 100 多条,而 snort 则为数千条.为了叙述方便,本节中用 m 表示规则集中正则表达式的数目,n 表示一条正则表达式的长度.为了完成对 m 条正则表达式的匹配,有两种策略可供选择:(1) 将每一条正则表达式编译成单独的一个 FA;(2) 将 m 条规则编译到同一个 FA 中.前者使用在 Snort 和 Linux L7 filter 中,后者则在文献[9,10]中被提出.

前面提到,对正则表达式的匹配需要使用有穷自动机来完成,但是 NFA 和 DFA 在实际应用中却有不同的优点和缺点.NFA 的优点是空间复杂度比较低,因为 NFA 的状态数目与正则表达式的长度成线性关系.然而在处理每一个字符时,由于必须逐个处理活动状态集合中的多个状态,因此匹配效率非常低.若将多条规则编译到同一个 NFA 中,虽然可以在处理过程中同时匹配所有正则表达式的公共前缀,但在实际应用中却会形成更大数量的活动状态集合,处理一个字符的时间复杂度和将每个正则表达式编译成单独一个 NFA 的时间复杂度相同.相比之下,虽然 DFA 处理一个字符只需要访问一个状态,但若将每条正则表达式编译成单独的 DFA,其时间复杂度同样将随着规则数目的增多而增大.而将所有的正则表达式编译成一个混合 DFA 时,则会导致其空间需求大大增加,以当前的硬件条件将无法满足如此大的内存需求.使用不同的方法和策略进行匹配时所需要的空间复杂度和处理一个字符的时间复杂度见表 1.

Table 1 Time and space complexity of various categories and methods

表 1 不同策略和方法下的时空复杂度

	Compile m regular expressions into m FA		Compile m regular expressions into a single FA	
	Time complexity	Space complexity	Time complexity	Space complexity
NFA	$O(n^2m)$	$O(mn)$	$O(n^2m)$	$O(mn)$
DFA	$O(m)$	$O(m2^n)$	$O(1)$	$O(2^{nm})$

从表 1 中可以看出,两种方法以及两种策略所形成的 4 种组合都无法同时满足空间复杂度和时间复杂度比较低的需求.要想对正则表达式进行实际应用,必须降低 NFA 处理一个字符所需要的时间复杂度或者对 DFA 所需要的内存空间进行缩减.NFA 的时间复杂度是由其理论模型所决定的,所以在不改变处理器体系结构的情况下很难对其进行改进,因此,当前的研究大多集中于对 DFA 的内存进行缩减.DFA 的内存消耗主要用于存储各个状态及其转换表,其空间需求由状态数目决定.下面给出几种典型的情况,它们会导致 DFA 状态数目急剧增长而又在实际规则中频繁出现.

1.2.1 单条正则表达式构建 DFA 时的状态膨胀

第 1 种导致状态增长的情况是,带有^符号的表达式中某个字符后面带有重复标记,且这个字符与其前驱字符出现了重叠覆盖.此时,需要大量的状态对所有可能的字符排列组合情况进行区分.

例如,表达式“^B+[^\$n$]{3}$D$”中,[^\$n$]与字符 B 的交集为 B,其构成的 DFA 如图 1 所示.一般来讲,如果重复次数为 j 次,则阴影部分所包括的状态所形成三角形高为 j+1,长为 j+1,总共需要的状态为 $O(j^2)$.

Fig.1 A DFA for pattern "^*B*+[^*n*]{3}*D*"

图 1 模式"^*B*+[^*n*]{3}*D*"形成的 DFA

第 2 种导致状态增加的原因是,不带有^符号的表达式通配符或者一组字符后带有有限重复操作符({*m*,*n*},{*n*}).此时,DFA 需要考虑任取字母表中的 *n* 个字符可以组成的所有情况,因此需要 $O(2^n)$个状态来记录.例如,**A*.{*n*}*CD*,如果在后续的字符串中出现了 *A*,那么如图 2 所示,每个状态都需要处理(*A* 和 not *A*)两种情况.为什么虽然第 2 种表达式与第 1 种表达式形式类似却有截然不同的空间复杂度($O(m^2)$和 $O(2^n)$呢?因为在第 1 种情况下,中间的状态每个只需要派生出一种新状态,或者转回到某个旧状态;而在第 2 种情况下,每个中间状态都可能派生出多个状态,从而造成指数级增长.

Fig.2 A DFA for pattern ".**A*.{2}*CD*"

图 2 模式".**A*.{2}*CD*"形成的 DFA

1.2.2 多条正则表达式构建 DFA 时的交互作用

上面描述了单条正则表达式构建在 DFA 时导致状态数目增长的情况.然而有些情况下,即使原来的单条规则不会引发状态数目的增加,一旦当将多条正则表达式编译在同一个 DFA 中,它们之间也会由于相互作用而造成 DFA 状态数目的急剧增长.例如,*ab*.**cd* 和 *ef*.**gh*,如图 3 所示,这两条规则在单独构建时都不会引起 DFA 数目的增长,但由于两个表达式中的".*"都可以匹配另一条正则表达式所匹配的所有字串,因此将它们编译到一个 DFA 中时,必须用额外状态记录所有可能的组合情况,形成的 DFA 和示意图如图 4 和图 5 所示.

一般而言,当多条规则编译在一起时,如果有 *k* 个正则表达式,每个表达式中出现 *x* 次".*",则需要 $O((x+1)^k)$个状态来记录所有可能出现的前缀的幂集.在这种情况下,即使每个规则只有 1 个".*"操作符,增加一个类似的规则也会使得 DFA 的状态数目增加 1 倍.在 L7-filter 中有很多类似的规则,例如,识别远程桌面协议的".*rdpdr

.*cliprdr.*rdpsnd”和 Internet radio 协议的“.*membername.*session.*player”.snort 中也存在大量类似规则,而且某些规则中,“.*”符号的数目甚至达到了 6 个.

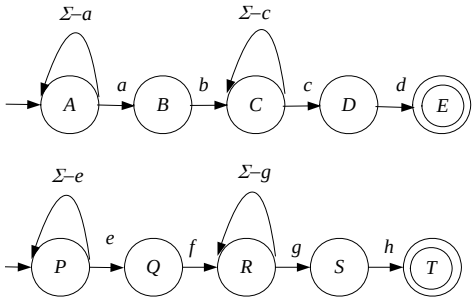


Fig.3 DFAs for pattern *ab*.**cd* and *ef*.**gh*

图 3 模式 *ab*.**cd* 和 *ef*.**gh* 各自形成的 DFA

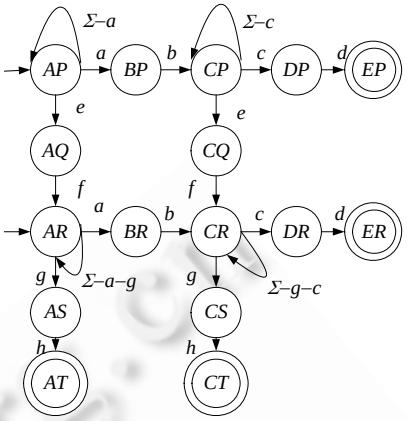


Fig.4 Composite DFA for pattern *ab*.**cd* and *ef*.**gh*

图 4 模式 *ab*.**cd* 和 *ef*.**gh* 编译在一起形成的 DFA

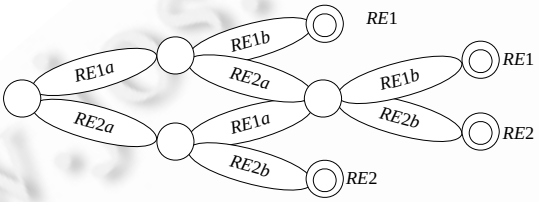


Fig.5 Exemplification of DFA obtained by compiling “.**RE*1*a*.**RE*1*b*” and “.**RE*2*a*.**RE*2*b*” together

图 5 将模式“.**RE*1*a*.**RE*1*b*”和“.**RE*2*a*.**RE*2*b*”编译在一起时的 DFA 结构示意图

本节叙述了一些能使 DFA 状态发生巨大膨胀的正则表达式操作符,而这这些操作符也正是在实际中最常用的.截止到 2007 年 11 月,8 536 条 snort 规则中有 5 549 条正则表达式.在这些表达式中,含有“.*”的规则有 905(16.3%)条,而含有“.{*n*}”操作符的规则有 2 445(44%)条.通过上述分析可知,将这些规则构建成 DFA 需要极大的内存空间,因此,基于 DFA 的匹配方法无法直接应用在实际中.

2 基于冗余消除的内存空间缩减

一个 DFA 状态通常由两部分组成:表示δ函数的转换表和接受规则列表.前者为一个大小为$|\Sigma|$的数组,在实际应用中,Σ通常选取为 ASCII 表,其大小为 256 项,每一项表示当前状态在接到一个字符输入时将跳转向哪个状态.后者通常为一个链表,表示匹配过程中若经由一系列转换跳转到了当前状态,则当前输入的字符串与哪些正则表达式匹配.这个列表可以为空,也可能为多项.如果某个状态的接受列表不为空,则称其为接受状态.一个具有 n 个状态 DFA 的主要存储消耗为一个 $n\times|\Sigma|$的二维表,它占用 DFA 空间的 95%以上.本节介绍的方法都是对这个二维表的存储空间进行缩减.

2.1 减少字母表的大小

对一个 DFA 来讲,转换表为一个二维表,其行数等于 DFA 的状态数目,而列数等于字母表中字符的个数.如果可以减少整个转换表的列数,则可以有效地降低内存消耗.基于寻找等价类的字母表压缩法就是利用这一原理来减少 DFA 的内存消耗.其基本思想是,字母表Σ中的字符可以分成若干类 $C_1,...,C_k$,每一类包含 1 个或多个字符.如果其中一类包含多个字符,那么这个类中的任意两个字符 c_i 和 c_j 必须满足:对于给定 DFA 的所有状态,都

有$\delta(s,c_i)=\delta(s,c_j)$.陈曙晖等人在文献[16]中提出一种基于集合交割的方法来对输入字符进行预编码,从而减少自动机转换表的列数.文献[11]中也提出了类似的方法,它们首先将整个字母表看成属于同一等价类,然后对自动机的状态进行遍历,根据每个状态的转换表不断将现有的等价类进行分割.文献[17]中进一步扩展了这种方法.文献[18]的作者则观察到,目标为同一个状态的转换,其对应的字符往往都是相同的.因此,它将状态和输入字符进行联合编码,形成 state-char 编码,进一步提高了对字母表中字符的压缩效果.

2.2 缩减单个状态的冗余转换

DFA 结构类似于一个有向图,它的每一个状态都可以在接受一个输入字符后跳转向另外一个状态.但是对于不同的字符,其跳转的目标状态却可能是相同的,这就形成了单个状态内的转换表项冗余.图 6 中给出了一个简单的 DFA 以及状态 1 的转换表,假设字母表$\Sigma=\{a,b,c,d\}$,可以看出,状态 1 的跳转表中 *b* 和 *c* 两项的值是相同的,因此可以通过消减这种相同表项的冗余来减少跳转表需要的空间.文献[11]中提出了一种增加间接转换表的方法来消除冗余,其基本思想是,将原始的表改用两个表来表示:一个表用来存储跳转的目标,称为转换表,但是对于多个相同的表项只存储 1 次;另一个表有$|\Sigma|$项,对应字母表中的每一个字符,但是它不存储实际的跳转目标,而是存储跳转目标在转换表中的索引,称为索引表.在实际应用中,跳转目标通常为整数或者指针,且最多有$|\Sigma|$种不同的值,因此索引表只需要用 log$|\Sigma|$bits 就可以对转换表进行索引.当原始表项中存在大量冗余时,使用这种索引表的方式就可以节约大量的空间.例如在图 3 中,假设原始表项为 32bits,则索引表的每一项可以用 2bits 来表示.由于有两个跳转项相同,因此可以节约 32×4–32×3–2×4=24bits.

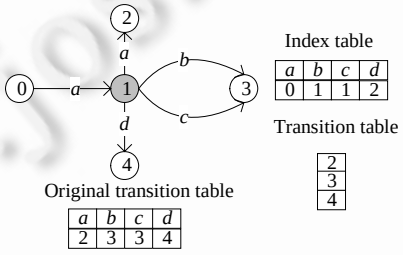


Fig.6 DFA and its original transition table, transition table and index table

图 6 DFA 及其原始转换表和索引表

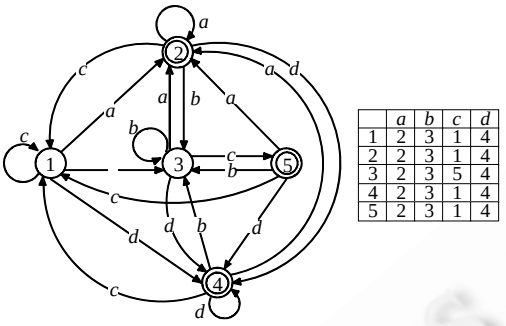
在实际的 DFA 中,多数状态往往只对少数几个字符有不同的目标状态,而大多数字符都会跳转向初始状态s_0或者 DFA 所形成的有向图中的某些起连接作用的关键节点,因此会存在大量的冗余项,使用索引表的方式可以节约大量的存储空间.然而这种方法在处理一个字符时,需要先访问一次索引表,然后再从转换表中找到真正的目标状态,比直接读取原始跳转表增加了一次访存操作.

2.3 消除不同状态间“等价”项的冗余

在 DFA 中,不但单个状态的转表内存在着冗余项,不同的状态之间也存在着大量的相同项冗余.图 7 给出了这种情况的一个典型例子.如图 7 所示的 DFA 由“*a*+”,“*b*+*c*”和“*c***d*”这 3 个正则表达式构建而成,共有 5 个状态,并且任何一个状态的转换表内部都不存在相同项的冗余.然而通过观察其原始转换表可以发现,在各个状态间却存在着大量的相同项冗余.为了描述方便,给出如下定义:两个不同的状态 *s* 和 *t*,如果对于相同的输入字符具有同样的跳转目标,则称这两个跳转项为等价项.如果可以消除不同状态之间由等价项引起的冗余,则可以进一步减少存储空间.

2006 年,Kumar 等人在文献[10]中首先提出了消除等价项冗余的方法,其基本思想是:如果两个状态 *s* 和 *t* 对于相同的字符具有相同的跳转目标,则在其中一个状态 *s* 中去掉所有和 *t* 中等价的表项,然后再引入一条从 *s* 到 *t* 的转换,称为缺省转换(default transition),它不需要输入字符也能进行跳转.在进行匹配时,如果访问到了状态 *s*,并且 *s* 对当前的输入字符 *c* 没有定义跳转目标,则根据其缺省转换跳转到 *t*,然后从 *t* 中取出跳转目标或者

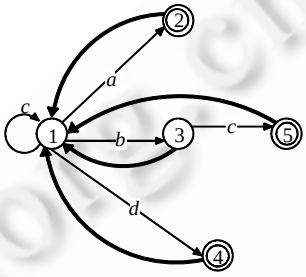
继续到 t 的缺省目标状态中去找跳转目标.这种为每个状态增加了缺省转换的 DFA 称为 Delayed Input DFA(D^{2}FA).D^{2}FA 的构建方法为:首先以 DFA 的状态为顶点形成一个完全图,然后计算任意两个状态间具有的等价项的数目,并以此数目作为完全图中每条边的权值;接着使用克鲁斯卡尔算法生成 1 个或多个最大生成树,称为消减树;最后以消减树的边作为缺省转换,完成冗余项的消除.图 8 给出了图 7 中 DFA 对应的 D^{2}FA 及其转换表,其中,状态 1 为整个消减树的根节点,加粗的黑线表示缺省转换,跳转表中空白项为消减掉的项,有值的转换项称为有效转换.可见,D^{2}FA 中的转换项数从原始 DFA 的 20 项缩减为 9 项.匹配时,若当前活动状态为状态 3,而输入字符为“d”,则由于状态 3 对应于“d”的跳转项已经被缩减,因此必须沿着其缺省转换,读取状态 1 的转换表并取得对应于字母 d 的跳转目标:状态 4.



	a	b	c	d
1	2	3	1	4
2	2	3	1	4
3	2	3	5	4
4	2	3	1	4
5	2	3	1	4

Fig.7 DFA and its original transition table

图 7 DFA 及其原始转换表



	a	b	c	d	Default
1	2	3	1	4	
2					1
3			5		1
4					1
5					1

Fig.8 D^{2}FA and its reduced transition table

图 8 D^{2}FA 及其缩减后的转换表

虽然 D^{2}FA 结构可以消除不同状态间等价项带来的冗余,极大地降低了 DFA 需要的空间,但是它增加了处理一个字符的时间复杂度.因为在原始的 DFA 中,每一个转换都会“消耗”一个字符,而 D^{2}FA 中,缺省转换是不消耗字符的.对于一个字符,D^{2}FA 可能需要经过多次缺省转换才能找到其跳转目标.为了能够在消减空间消耗的基础上保持匹配效率,Kumar 在文献[19]中对 D^{2}FA 进行了改进,提出了 CD^{2}FA.CD^{2}FA 在转换项中不再直接存储跳转的目标状态,而是存储以下信息:(1) 从消减树根节点到状态自身的路径所经过的祖先节点;(2) 这些祖先节点的有效转换;(3) 当前节点所拥有的有效转换;(4) 路径中所经过的接受状态节点.因此在 CD^{2}FA 中,转换项被称为 content label.同时还提出了一种构造哈希函数的方法,这个哈希函数可以用 content label 为输入,计算出对应于每个字符的跳转目标的地址.使用 content label 可以使 CD^{2}FA 能够和原始 DFA 一样,在处理一个字符时仅访问 1 个状态.但是由于每个 content label 项的容量有限,因此消减树的直径不能过大,通常取小于等于 2,这就影响了缺省转换的消减效果.Becchi 在文献[20]中提出了另一种简单而有效的办法:首先将原始 DFA 的每个状态附加一个深度值,这个值等于从初始状态 s_0 到其自身所需要经过的最短路径;然后在构造消减树时,只允许缺省转换从深度值较高的状态指向深度值较低的状态,但不需要限制消减树的半径.这种方法生成的 D^{2}FA 在处理某一个字符时可能会访问多个状态,但是可以证明,在处理一个长度为 n 的字符串时,其访问的状态数目不大于 2n.

2008 年,Ficara 等人在文献[18]中提出了另一种消除状态间由等价项而形成的冗余的方法.这种方法的思想源于 D^{2}FA,但是却没有使用缺省转换,而是为 DFA 增加了一个临时转换表(local set).他们观察到:在 DFA 中相邻的状态之间,一般会有大量的等价项.例如,在图 7 中,状态 2 和状态 1 具有完全相同的转换表,而状态 3 仅与状态 1 在输入字符 c 时具有不同的跳转目标.因此,如果每个状态仅仅存储与其相邻状态不同的表项,而其余表项直接从其父节点继承而来,则形成的 DFA 与原始 DFA 等价.这种仅仅存储与其上级节点不同的项而形成的 DFA 也因此被称为δFA.其构造过程为:首先申请一个新的二维转换表,然后从原始 DFA 的初始状态 s_0 开始,先将初始状态的转换项全部拷贝到新表中的对应行,再从这个状态出发遍历 DFA 中所有状态,如果一个状态对应某个字符转换项与其任意一个父节点的相应项不同,则需要在新表的对应项中存储这一项,否则丢弃此项.

图 7 中 DFA 形成的δFA 及其转换表如图 9 所示.可以看出,除了状态 1 和原始 DFA 具有相同的转换表之外,其余状态都只存储了 1 项有效转换,并且不需要引入额外的缺省转换.δFA 将原始 DFA 的 20 项转换缩减到了 8 项.在匹配时,δFA 需要一个临时转换表,其初始值为δFA 初始状态 s_0 的转换表,每次经过转换到达一个新状态时,首先根据新状态的转换表更新临时转换表的表项,然后再根据输入字符从临时转换表中找到目标状态.因此,处理每个输入字符只需访问 1 个状态.图 10 给出了图 9 中δFA 在处理字符串 *abc* 时的过程,圆圈中表示状态及其转换表,下面的表为临时转换表在遍历到相应状态时的内容.在初始状态 1 时,其内容与状态 1 的转换表完全一致;而通过字符 *a* 跳转到状态 2 时,它首先将对应于字符 *c* 的转换表项进行更新,然后通过 *b* 跳转向 3.此时,再次将对应 *c* 的表项更新成 5,然后再继续执行.δFA 并不能完全避免冗余,例如,图中状态 2 和状态 1 对应于 *c* 的跳转目标依然相同.这是因为对应于字母 *c* 的转换虽然与状态 1 相同,但是却与状态 5 不同,而在遍历中,状态 1 和状态 5 都可能成为状态 2 的前驱节点,因此状态 2 必须存储对应于字符 *c* 的转换,才能保证在各种情况下都与原始 DFA 等价.它还必须增加额外的操作来更新全局表项,如果一个状态存储的有效状态比较多,则更新操作将会降低匹配的效率.

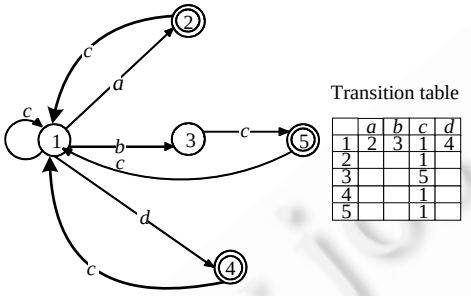


Fig.9 δFA of *a*+, *b*+*c*, *c**d*+ and its reduced transition table

图 9 *a*+,*b*+*c*,*c**d*+所形成的δFA 及其缩减后的转换

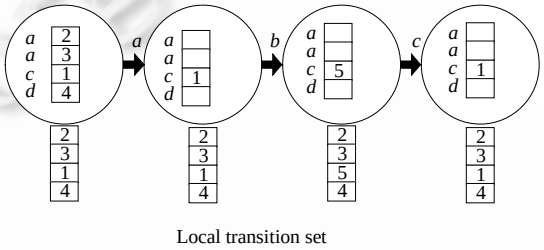


Fig.10 A lookup example of δFA

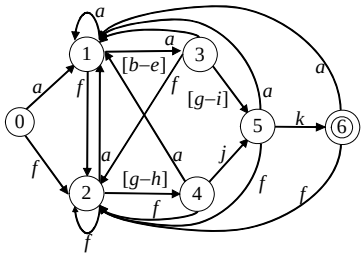
图 10 δFA 查找过程示例

2.4 消除不同状态间非等价项形成的冗余

上面介绍的方法都是消减不同状态间等价项形成的冗余,这意味着只有两个状态的转换表项对应的转换字符和目标状态都相等时才能进行消减.但还存在另一种形式的冗余:不同状态间非等价项的冗余.下面以图 11 为例说明这种情况.图 11 的左半部分给出了由正则表达式(*a*[*b*−*e*][*g*−*i*]|*f*[*g*−*h*]j)*k*+生成的 DFA,右边给出了状态 1~状态 4 的索引表和转换表.为了便于观察,图中省略了所有目标为状态为 0 的转换.可以发现,虽然在实际的转换表中仍存在冗余,但这些表项却不都是等价项,因此不能使用上面的方法来消除.例如,状态 3 和状态 4 中都有跳转向状态 5 的项,但却分别对应字符[*g*−*i*]和 *j*.

为了能够对这种冗余进行消除,2007 年,Michela 提出了状态合并(state merging)的方法[11].其基本思想是,如果两个状态具有目标相同的转换表项(无论是否对应于相同的字符),则将两个状态合并起来,形成一个混合状态.混合状态拥有合并前原始状态的两个索引表以及一个合并后的转换表.图 12 中给出了图 11 中 DFA 经过状态合后并得到的新的 DFA 及其对应的转换表.如图所示,状态 1 和状态 2 合并之后形成了混合状态 1_2,状态 3 和状态 4 合并之后形成了混合状态 3_4.在状态合并之后,还必须进行以下两个操作来保持新的 DFA 与原始 DFA 等价:(1) 在对状态进行合时,将原来的两或多个转换表合并成了 1 个,所以某些目标表项的索引位置可能会发生变化.例如,状态 2 的对应于字母 *g* 和 *h* 的项,在原来的索引表中为 3,而合并后,其索引值变成了 4.因此,在合并转换表后必须对混合状态的索引表进行更新,使其指向的目标转换项与原来的项一致.(2) 状态合并后,两个状态变成了 1 个状态,原来目标指向这两个状态的转换也都更新为指向这个混合状态.但是这会使得遍历到混合状态后无法决定选取哪个索引表来进行下一次跳转目标的选择,因此在状态合并后还需要对合并前的索引表进行标记.以图 12 为例,状态 3 的索引表成了标记为 3_4.0,而状态 3 的索引表则标记为 3_4.1,原来指向

状态 3 的跳转目标项也相应地添加一个标记 0,目标为 4 的项则增加一个标记 1.这样,原来跳转向状态 3 的项就可以根据标记 0 去状态 3_4 中的索引表 3_4.0 中去查找下一次跳转的目标.为清晰起见,图 12 中的转换表没有给出这个标记,而是在 DFA 示意图中的转换上标出.



Index table

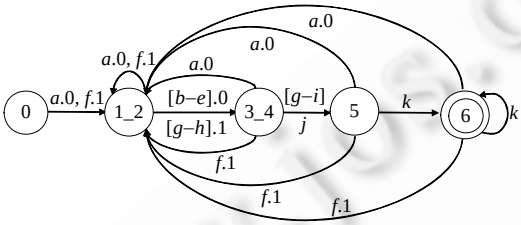
	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*
1	1	3	3	3	3	2	0	0	0	0
2	1	0	0	0	0	2	3	3	0	0
3	1	0	0	0	0	2	3	3	3	0
4	1	0	0	0	0	2	0	0	0	3

Transition table

1	0	1	2	3
2	0	1	2	4
3	0	1	2	5
4	0	1	2	5

Fig.11 DFA and its transition table and index table

图 11 DFA 及其索引表



Index table

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*
1_2.0	1	3	3	3	3	2	0	0	0	0
1_2.1	1	0	0	0	0	2	4	4	0	0
3_4.0	1	0	0	0	0	2	3	3	3	0
3_4.1	1	0	0	0	0	2	0	0	0	3

Merged transition table

1_2	0	1	2	3	4
3_4	0	1	2	5	

Fig.12 DFA after state merging and its merged transition table and index table

图 12 状态合并后的 DFA 及其合并后的转换表和索引表

状态合并与基于消除等价项的方法相比有两个优点:首先,它不需要被消减项是完全等价的,只要有相同的跳转目标就可以;其次,随着合并的进行,还可以创造出更多的合并机会,而消除等价项的方法是静态的.其不足之处在于:首先,它需要和 D^{2}FA 一样建立一个以 DFA 状态为顶点、顶点间公共转换数目为权值的完全图,因此当 DFA 状态为 n 时,需要 $O(n^2)$的额外空间.其次,在原文的方法中,每个混合状态最多可以容纳的状态数目是需要人为指定的(用 max_*labels* 表示),由于在每一次合并之后需要重新计算和合并后状态相关的权值,因此整个合并过程的时间复杂度高达$(1-1/\max_labels)n^3\log n$).同时,由于 max_*labels* 为人为指定,所以每个混合状态的转换表项数不相等,这会造成相应索引表项所用的 bit 位数也不相等.在实现中,要么使用额外的标记进行记录从而付出额外的空间代价,要么选取统一的长度(最长 bit 位数),这也会造成空间的浪费.后一种浪费的本质是对索引表项的索引能力的浪费.为此,文献[21]中提出了转换表共享方法以改进状态合并方法.转换表共享的基本思想是:对有相同转换项的多个状态不进行合并,而是让它们共享同一个转换表.和状态合并一样,它首先将多个状态的转换表合并,形成一个大的公共转换表,然后对共享这一转换表的状态的索引表进行更新,以保持与原始 DFA 的等价性.不同之处在于,它并不将这些状态合并成一个混合状态,而是保持状态的独立,因此不需要对转换和索引表进行额外的标记.更重要的一点是,它并不直接指定 max_*labels*,而是指定索引表项的 bit 数目 w,然后在共享的过程中使用启发式共享策略,逐个选择下一个状态,使得最后形成的公共转换表项恰好等于 2^w 项,这能够使得索引表项的表达能力被充分利用.考虑到 DFA 中相邻状态往往具有更多的等价转换,转换表共享可以从 DFA 的初始状态开始,按照宽度优先的顺序进行试探,从而省去装换表合并时对以状态为顶点的权值图的搜索和更新过程.

3 基于自动机结构改进的内存缩减

第 2 节所述的基于冗余消除的内存缩减技术可以大规模缩减 DFA 所需要的内存空间,甚至可以缩减掉原始 DFA 所占用内存的 90%以上.然而这类方法无法彻底解决 DFA 的空间占用问题:首先,DFA 的内存是随状态数目的增长而增长,从第 1 节的分析可知,DFA 的状态数目是平方级甚至指数级的增长,而无论怎样缩减,要想保持与原始 DFA 的等价性,都至少要保持其状态之间的连通性,这意味着最多将 DFA 的图形结构缩减成一棵以各个状态为顶点的树,而树的边与顶点是成线性关系的,因此冗余消除只是线性的缩减,无法解决状态的指数级增长带来的空间消耗;其次,基于冗余消除的缩减方法大都需要额外的辅助内存空间,例如,D^2FA 和状态合并方法都需要 $O(n^2)$空间来保存权值图.但有些规则集根本不能在现有的硬件资源下建立起单个的 DFA 结构,因为其原始 DFA 本身所需求的内存就已经超过了系统可提供的内存.此时,即使是δFA 和转换表共享这些方法可以在原始 DFA 上执行的方法,也不能处理这种情况.本节对另一种内存缩减方法进行分析和介绍,这种方法从改进自动机的结构出发,力图从根本上解决空间膨胀问题;同时,这种方法还可以和前面的方法叠加使用.

3.1 规则分组

从第 1.2.2 节可知,当多条正则表达式编译成一个 DFA 时,正则表达式之间由于相互重叠和影响,会导致 DFA 的状态数目大规模的增长,甚至多个本身不会引起状态“爆炸”的正则表达式,由于互相交互也会导致 DFA 状态剧增.针对这种现象,Yu 等人在文献[9]中提出了将正则表达式进行分组的思想.其方法是:首先计算正则表达式两两之间是否“相交”(引起状态增长),然后构造一个“相交”关系图.关系图以每一个正则表达式为顶点,如果两个表达式相交,则用一条边连接相应的两个顶点.在进行分组时,首先选择一条与其他表达式具有最小相关度的正则表达式开始,然后按照相同的原则向这个组里不断添加,直到这个组形成的 DFA 内存超过预先设定的阈值,再开始创建另一个新组.重复这个过程,直所有的表达式都被分配出去为止.这种分组尽可能地减少了表达式之间的交互作用,大大减少了空间需求.Yu 等人提出的分组方法是一种朴素的启发式分组,而文献[22]中则提出一种更加合理的分组策略:首先定义了膨胀率 DR 来描述正则表达式的空间膨胀特性;然后,基于 DR 提出一种分片算法,这种方法有效地选择出导致 DFA 状态膨胀的片段并进行隔离,从而降低了单个正则表达式存储需求.同时,文中还基于正则表达式的组合关系提出了一种选择性分组算法.

分组方法对 DFA 和正则表达式之间的关系进行了研究,但是只是做了简单的分组处理,通过将一个 DFA 分解成 k 个 DFA 的方法降低了空间需求.但在内存带宽不变的情况下,会将匹配效率降低为原来的 $1/k$.

3.2 混合结构自动机

NFA 的空间复杂度和正则表达式的长度呈线性关系,而 DFA 是通过 NFA 确定化得到的.从这一点出发,早期的研究中就出现了通过对 DFA 和 NFA 之间的转化程度进行调节来实现状态控制的基本思想[23,24].近年来,随着空间问题的日益凸显,文献[25]中提出了 lazy DFA 来匹配正则表达式.它只将命中频率较高的规则转化成 DFA,而对那些不常出现的规则,先维持它们的 NFA 状态,如果在匹配过程中遍历到了相应的 NFA,再将其确定化成 DFA.这样,lazy DFA 通常比全部确定化的完全 DFA 具有少得多的状态,且其平均时间复杂度与完全 DFA 相当, Bro 中就使用了这种方式.但其最坏时间复杂度比较差,因为对于第 1 次被命中的规则,它需要首先将其进行确定化,这就导致了对这条数据处理的速度会很慢.并且这种动态扩展的方法只适合软件实现,如后面将叙述的,现在的具体实现可能还需要一定的硬件结构来进行加速.

文献[13]中提出了一种混合自动机的方法,其基本思想是,在将整个规则集编译成一个 NFA 结构之后,并不对它进行完全的确定化,而是在确定化的过程中判断状态之间的跳转是由哪种情况引起的,如果一个跳转是由第 1 节中分析的几种导致状态增长的情况形成的,就停止确定化.停止确定化的状态称为边界状态.进行部分确定化的结果就是形成了一个混合的自动机结构,它的前面一部分是 DFA 的状态,而在每个边界状态之后都带有一个 NFA,这个 NFA 以边界状态作为初始状态.混合自动机具有以下性质:(1) 初始状态是 DFA 状态;(2) NFA 部分只有在匹配过程访问到边界状态时才会被激活.最重要的一点是,不存在从 NFA 向 DFA 部分的转换,因此可以互不影响的执行.图 13 左半部分为混合自动机的示意图,head-DFA 表示被确定化了的状态组成的

DFA,有阴影的小圆表示边界状态,它们通常是由“.*”,“.{*n*}”等符号形成的.其后跟着若干个 NFA,称为 tail-NFA.

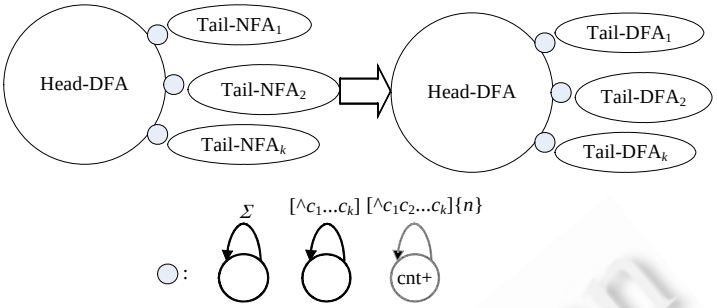


Fig.13 Exemplification of hybrid FA

图 13 混合 FA 示意图

混合自动机可以有效地控制由于确定化带来的状态增多,但当 NFA 部分频繁地被激活时,其处理速度退化得也很快.为此,文献[12]中提出了两种改进措施.第 1 个措施是,将尾部的 NFA 也确定化成 DFA,这样可以保证每个尾部的 DFA 只有 1 个状态处于激活.不过,这种方法只对“.*”类型边界状态后的 NFA 有效才有效,因为如果是对[^*adf*]这种规则,则可能需要维持多次重叠激活才能保证匹配的正确性,而“.*”却只需要 1 个.第 2 个措施针对“.{*n*}”类型的操作符,在构建自动机时,它将产生多个相同的状态,为了减少状态数目,可以用一个状态并附加计数器的形式来实现对重复次数的记录.根据不同的情况,可以使用 1 个或者多个计数器来,对多个计数器用存储差值的形式进行实现,这样,只更新两端的计数器就可以管理整个计数器组.

3.3 基于历史记录的自动机

虽然混合自动机兼具 DFA 的速度和 NFA 的空间复杂度,但是在实际应用中,尤其是规则集已知的情况下,攻击者很容易构造出特殊的数据包,使得其 tail-FA 部分不断被激活,从而使整个自动机的效率接近于 NFA.为了改进这种情况,文献[12]中提出了一种基于历史记录的自动机(H-FA),这种自动机在避免状态膨胀的同时,任一时刻都只有 1 个活动状态.这个方法利用了第 1.2.2 节中的分析:DFA 的状态膨胀通常是由于某些规则的形式为多个简单特征被闭包操作符连接起来(如“.*”或者“[*az*]*”),而一个输入字符串可能同时匹配多个简单特征,此时,DFA 必须增加状态来模拟这些特征的排列组合情况,从而导致了 DFA 状态的指数级增长.为了避免这种指数级增长,一个直接的办法就是构建一种可以记住更多信息的自动机.NFA 具有这种能力,因为它的活动状态是一个集合,可以记录每一条特征的匹配进度.但是,NFA 处理一个字符的时间复杂度太高,无法满足匹配效率的需要.H-FA 的基本思想是增加一个很小的高速缓存空间(cache),称为历史缓存.将构建自动机时的关键事件记录在这个 cache 中,例如遇到一个闭包符号,这个思路类似于前面的边界状态.由于状态的膨胀是由于闭包符号引起的,所以一旦在 cache 中记录了这个事件,就可以依靠读取这个事件的记录来避免使用多个状态来表示其排列组合情况.

H-FA 的结构类似于普通的 DFA,但是它的每一个状态对一个字符可能会有多个转换表项,但在匹配过程中每次只会选取其中 1 个,而选取哪一个则依赖于 cache 中的内容.因此,H-FA 进行匹配必须依靠历史缓存(cache)中的记录,且 Cache 中记录的内容也会随着匹配的中间结果不断被更新.H-FA 的大小取决于如何选择 cache 中的存储内容,只要策略得当,它就可以消耗非常小的内存空间.

图 14 中给出了一个 H-FA 的例子.图中首先给出了一个 NFA,然后对这个 NFA 进行确定化形成 DFA,以及一个对应的 H-FA.可以看到,在原始的 DFA 中有 10 个状态,而 H-FA 中有 6 个状态和一个 *flag* 标记.这个 *flag* 有两种状态:set 和 unset(用<=1 和<=0 表示),由字符 *a* 引起的从状态 0 到状态(0,1)的转换会引发 reset flag 的操作,而由字符 *b* 引起的从状态(0,1)到状态 0 的转换会引发 set flag 的操作.有一些转换需要依赖 *flag* 的状态进行选择,例如,当状态 0 收到输入字符‘*c*’时,如果 *flag* 处于 set 状态,则会跳转向状态(0,3);否则,将会跳转向状态 0 本

身.可以验证,这个带有 *flag* 标记的 H-FA 与原始的 NFA 接受同样的语言,但是它在处理一个字符时,仅需要访问 1 个状态.

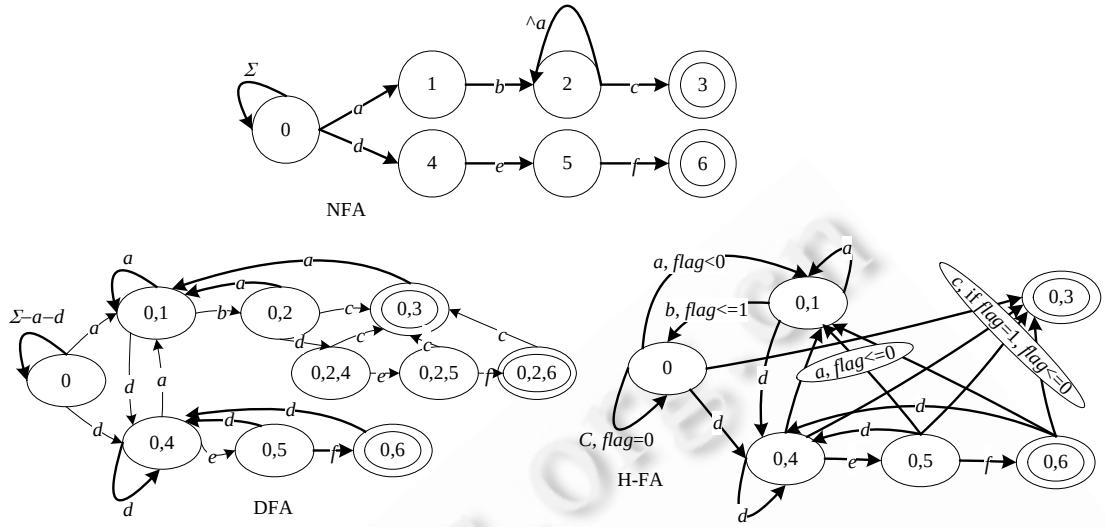


Fig.14 NFA and its corresponding DFA and H-FA

图 14 NFA 及其对应的 DFA 和 H-FA

3.4 XFA

H-FA 拥有对一个输入字符仅访问 1 个状态的特性,但是其状态对于一个输入字符却可能有多个跳转项.在需要存储的内容比较多的情况下,某些状态可能会出现大量的条件转换.同时,H-FA 只是对闭包符号带来的状态膨胀进行了改进,而无法处理另一类指数级增长的符号(例如,“.{*n*}”).Smith 等人在文献[14,25]中对这种使用额外标记进行中间结果记录的方法进行了改进和扩展,提出了 XFA 结.和 H-FA 一样,XFA 也使用了一个高速的数据区来存储信息,但是它在其中存储了更多的信息,除了 H-FA 中使用的标记以外,它还专门为“.{*m*,*n*}”,“.{*n*}”型操作符设计了计数器型标记.这个计数器类似于混合自动机的优化策略中提到的计数器,可以对某个字符或字符字串出现的次数进行记录.除了标记之外,还在数据区中存储了操作代码,用来实现对标记进行更新.它将这些操作和状态关联起来,当匹配过程中访问到某个状态时,就执行与这个状态相关的代码,同时根据与这个状态关联的标记来判断这个状态是否为接受状态.XFA 与 H-FA 的一个重要不同之处在于:它的状态对于一个字符,只有 1 条转换边,而不是存在多个条件转换边.

图 15 中给出了针对两种不同操作符的 XFA.图 15(a)中的上图给出了正则表达式“*ncmd*[^*n*]{200}”生成的 DFA,可以看到,为了记录[^*n*]{200},原始 DFA 使用了 200 个状态,而其下面的 XFA 中使用了一个变量 *c* 以及相应的“赋值”、“reset”、“自减”以及“条件判断”操作避免了 200 个重复状态的使用.同样,图 15(b)中的上图给出了两个 XFA,分别是从正则表达式“*ab.*cd*”和“*ef.*gh*”构建而成,它们分别使用了 $flag_1$ 和 $flag_2$ 来标记是不是已经匹配了各自表达式的前半部分.对比第 1 节中的例子可以看出,对于这种类型的规则,单独生成 DFA 和 XFA 其状态数目是相同的,反而是 XFA 比 DFA 多了一个标记和若干个操作.但当多条这种类型的正则表达式编译在一起时,XFA 就有了明显的优势,因为原始 DFA 需要 17 个状态,而 XFA 仅需要 7 个.因为有了标记变量和操作,XFA 就不需要更多的状态来记录两个表达式前缀匹配的排列组合情况,避免了 DFA 中状态数的膨胀.由于 XFA 在匹配过程中每次经过一个状态时都可能要执行多条操作指令来更新各种标记,Smith 等人还提出了一种优化技术,将构建后 XFA 的数据区映射为一种称为 efficiently implementable data domain(EIDD)的结构,以便提高匹配的效率.

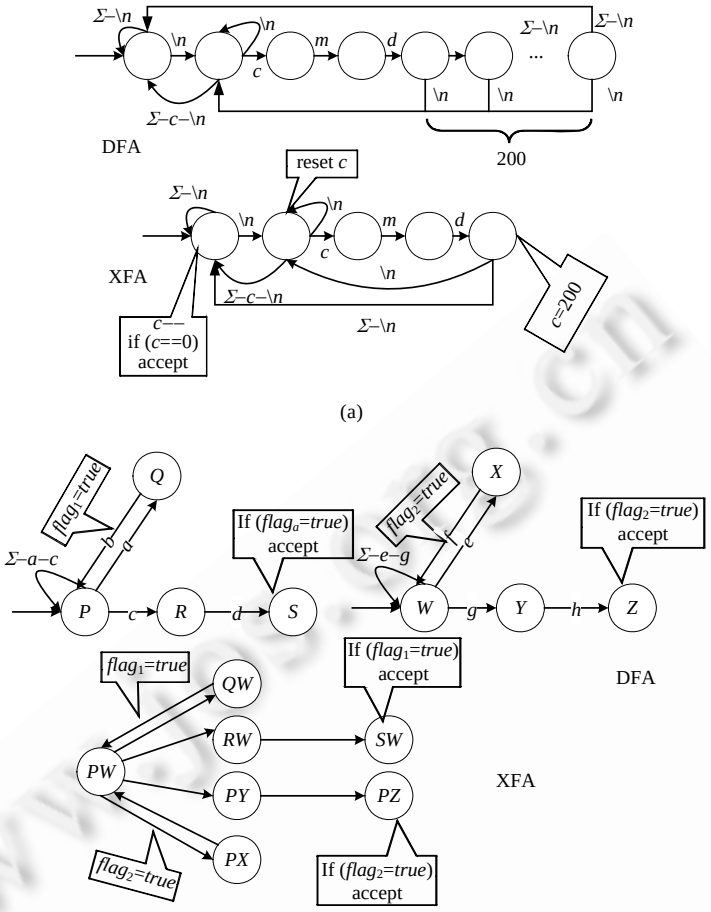


Fig.15 Original DFAs and its corresponding XFAs

图 15 原始 DFA 及其对应的 XFA

4 基于不同体系结构的实现优化

第 2 节和第 3 节分别介绍了两类内存缩减技术,从而使基于自动机的匹配方法可以在当前的硬件资源上运行.但这两种方法有一个共同的缺点:内存的缩减是以牺牲部分时间复杂度换取的.基于冗余消除的内存缩减,会使得自动机在处理一个字符时需要访问多个状态或者多次访存才能得到真正的跳转目标,而改进自动机结构的方法则需要在跳转的同时增加额外的操作来辅助匹配过程的正确执行.这些操作或者增加了访存次数,或者增大了计算量.然而,当前多种多样的计算架构为消除这种负面影响提供了可能.例如,多级内存结构可以缓解访存延迟,多线程并行能力可以实现多个引擎同时工作或同时对多个跳转项进行查询和更新缓存内容等等.为了能够实现尽可能大的吞吐率,研究者在不同体系结构下的实现上提出了各种优化办法.

目前,用来进行特征匹配的硬件结构可以大致分为两类:可编程逻辑门阵列(field programmable gate array,简称 FPGA)和以内存为中心的计算体系,如图 16 所示.FPGA 的优点是可重塑性强,具有较好的并行执行能力,能够在 1 个时钟周期内完成多步计算,因此有很多特征匹配的研究都是基于 FPGA 的特征匹配[26−29].其缺点在于时钟频率比较低,计算速度慢,而且价格昂贵,资源(空间、功耗)消耗大,扩展性差,因此用来进行正则表达式这种资源消耗较多的匹配时具有很大的局限性.而以内存为中心的计算架构,比如通用 CPU,ASIC,或者 FPGA/

ASIC+内存等,则具有计算速度快、通用性和扩展性比较好等特点.实验结果表明,基于 NFA 的正则表达式匹配可以在 FPGA 上取得更好的最好的吞吐率.而在以内存为中心的架构下,DFA 和混合自动机更加有优势[30].下面介绍一些在特定体架构下的优化技术.

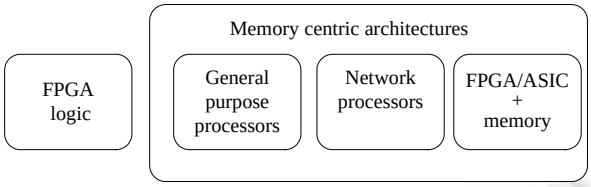


Fig.16 Various computing architectures

图 16 计算架构分类

在基于 FPGA 的研究中,主要针对的问题是减少存储空间和优化电路.由于不是本文讨论的重点,在此只进行简要介绍.文献[31,32]都是通过共享规则间的相同前缀来减少 FPGA 所需要的电路总数.文献[31]中给出了一种算法来生成共享匹配串前缀的电路,而文献[32]则提出了使用多路选择器对多个共享前缀子串的表达式进行并行匹配.文献[33]中针对正则表达式中有限次重复操作符(如 $a\{1,8\}$)的匹配进行逻辑电路的优化设计,以减少存储空间的需求.文献[34]中提出了一个基于 FPGA 的 NFA 匹配引擎,它将每个状态编码成一个触发器,可以在 1 个时钟周期里处理 1 个甚至多个字符而不受活动状态数目的影响,但其效率会随着每个状态拥有的转换数目的增多而下降.文献[35]则提出了一种将 NFA 转换为 VHDL 并对其进行优化的方法.

下面重点讨论一组在以内存为中心的架构下的实现技术.文献[11]针对 D^2FA 提出了一个基于拥有多个嵌入式内存的架构的实现策略,并设计了一种算法可以有效地将 D^2FA 映射到这个体系结构上.这种拥有多个等量内存块的架构是一种逻辑结构,它可以用带有 Mb 级别的 FPGA(如 Xilinx Virtex-4[36])、有片上内存的 ASIC 以及通用 CPU 来实现.使用片上内存来存储自动机而不是直接将其固化到逻辑门中,是因为考虑到正则表达式的规则集可能会经常发生变动,因此整个系统需要有更好的灵活性和可扩展性,并且嵌入式内存块越小,其访问的时钟频率通常就越高.因此,将 D^2FA 映射到内存中时,要尽量减少分片率并且每片内存都有相等的访问率,这样整个系统中就不会出现效率瓶颈.在这个逻辑架构中,每个内存可以被一组正则匹配引擎访问,每个引擎可以独立处理数据包,多个引擎可以用来实现包级别或者流级别的并行策略加速.由于对包的处理速度取决于单块内存的带宽,因此整个系统的吞吐率接近于所有内存的带宽和.

文献[22]为 XFA 提出了一个芯片级的架构设计.这个架构可以在支持 24 567 个 XFA 状态时拥有 10Gbps 的吞吐率.它有 8 条包处理流水线,每个流水线分为 3 部分:匹配引擎、代码查询引擎和处理单元,处理单元的时钟频率为 500MHz.匹配引擎中使用多端口的 SRAM 来存储转换表,并使用了第 2 节所述的基于 D^2FA 的内存缩减技术,每个 state id 用 15bit 来标记.引擎处理每个字符时,需要访问一次转换表,每两个 cycle 处理一个字符.代码查询引擎的输入是匹配引擎所遍历的状态序列,它将这个状态序列转换成一系列需要执行的操作指令.处理单元执行代码查询引擎给出的指令序列.被执行的指令放在一个 64K 的指令缓存中,由于指令缓存不足以存储 XFA 中所有的指令,因此启用了一些缓存更新策略,在遇到新指令时,首先将指令拷贝到缓存中再执行.每个处理单元使用 32/16bit 的寄存器来存储变量,操作指令不需要访存或者分支判断,所有的指令都是 2 个字节.此外,还有一个调度器辅助处理器在多个处理流之间进行切换.对于匹配引擎来说,切换仅仅需要更换一次当前的活动状态,而处理单元则需要保存好寄存器现场,然后读入新的现场.为了不减慢处理单元的速度,使用两组寄存器和相关部件,处理单元在完成当前任务的同时,另外一组设备中的现场可以被写入到缓存中或从缓存中恢复其他现场.

文献[37]中提出了一种针对混合自动的逻辑架构和实现策略.在这个架构中,head-DFA 和所有的 tail-Fas (DFAs/NFAs)由两个线程分别执行.因为 head-DFA 始终处于活动状态,且每个输入字符都需要先经过它的处理,不带有“.*”,“.{n}”等符号的正则表达式也会在 head_DFA 中直接匹配成功,因此 head-DFA 部分必须是可快速访

问的.而 tail-FAs 则只有在边界状态被访问后才会被激活,并且在一段时间后会失去活动状态,因而可以被离线存放,待触发时再加载.将 head-DFA 和 tail-FA 存储在不同的内存区域,就可以用两个线程并行执行而不发生访存冲突.

在这个架构中,还开辟了一个专门的内存区域,用来存放对混合自动机进行优化时使用的计数器.所有的 DFA 都使用了文献[20]中所述的缩减技术,访问一个状态仅需要 1 次访存.匹配时,输入字符首先由 head-DFA 引擎进行处理,如果某个字符串在处理过程中访问到了边界状态,则将一个任务放入称为 activation FIFO 的队列中,这个任务本身包含了边界状态、字符在相应字符串中位置等信息,Tail-FA 的引擎可以通过读取这些信息来完成边界状态所激活的自动机的匹配流程.

除了上述工作以外,Michela 等人还在文献[38]中提出了 Intel IXP 2800 NP 处理器下的 NFA、DFA 以及混合自动机的实现策略,并与两种通用处理器的结果进行了比较.这些策略的一个特点就是用 IXP 的多线程处理能力和异步内存操作来消除内存延迟,加快处理速度.Hayeng 等人在文献[39]中提出了一个使用 GPU 实现 XFA 的方案,利用面向数据流的处理特性来加快匹配效率.

5 总结和展望

本文首先阐述了特征匹配的发展现状,说明了正则表达式作为特征描述语言的巨大优势;然后对正则表示在深度包检测中的匹配方式以及方法进行了介绍,并着重强调并分析了当前的匹配方法所面临的挑战;最后对近年来各个研究团体为了能使正则表达式真正实用化所做的工作及发展脉络进行了详细的分类阐述.

不难看出,本文中介绍的所有方法都有一个共同的特性:以时间换空间.无论是增加索引表、引入缺省转换,还是使用各种标记来辅助记录匹配过程中的中间结果,其本质都是通过增加计算量来减少内存空间的使用.虽然可以借由各种架构来优化不同方法的效率,但在不远的将来,这些方法都将无法应对网络中攻击和应用协议种类不断增多所带来的挑战.因为从直观上来讲,精确字符串、安全类应用需要的规则以及正则表达式之间的关系可以用图 17 来表示,正则表达式的空间消耗问题正是由其本身的表达能力所带来的,它的描述能力过于强大,以至于它所能描述的问题空间在某些情况下超过了安全类应用的范围.例如,“.*”符号在理论上可以是任意多个字符,但是在实际中的每次输入都是有确定长度的.当然,还有一些规则无法用正则表达式方便地表示,比如,协议的两个字段之间满足一定的代数计算关系(packet[1,2,3,4]=packet[5,6,7,8]^2,其中,packet[5,6,7,8]和 packet[1,2,3,4]表示 4 字节的整数),如果一定要用正则表达式表示,则需要穷举 2^{32} 种可能,所需的正则表达式将非常长,所以不适合用正则表达式表示.

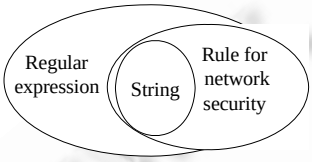


Fig.17 Relations of different rule description languages

图 17 不同规则描述语言之间的关系

若要更好地解决复杂特征匹配问题,可以从以下两个方向进行努力:

(1) 进一步发展第 3 节中提出的思想:改进基于自动机的匹配模型,将自动机对逻辑关系的判断以及各种高效的精确串匹配结合起来,充分挖掘面向网络安全的应用中规则的特点和匹配行为的特点来发挥经典多模串匹配算法的高效性和自动机模型对复杂关系的判断能力.例如,面向网络安全的复杂规则主要是用来描述攻击行为的多个阶段或某个恶意代码的多个关联特征,因而具有分段性;而匹配动作在整个处理过程中则表现为一个频繁而短暂的过程,并且只有少数数据包才会命中特征,绝大多数都不会命中任何特征.这就使得快速的简单特征匹配和相对慢速的完整特征验证可以分别用不同的匹配模型来完成,从而形成内存需求和处理速度的平衡.

(2) 寻找更好的规则描述方法来对安全问题进行恰当的描述.这种规则可能是正则表达式的一个子集的扩展,也可能是对字符串型规则的扩展,然后再找出针对这种特定规则的高效匹配方法.近年来,已经有研究人员在这方面进行了相关的工作,如曹京等人提出的布尔表达式[40,41],就是能力处于字符串和正则表达式之间的一种描述方法.

References:

[1] Snort 2.8.x. 2009. http://www.snort.org

[2] Aho AV, Corasick MJ. Efficient string matching: An aid to bibliographic search. Communications of the ACM, 1975,18(6): 333−340. [doi: 10.1145/360825.360855]

[3] Wu S, Manber U. A fast algorithm for multi-pattern searching. Technical Report, TR-94-17, Tucson: Department of Computer Science, University of Arizona, 1994. 1−11.

[4] Allauzen C, Raffinot M. Factor oracle of a set of words. Technical Report, 99-110, Institute Gaspard-Monge, University deMarne-la-vallee, 1999. 1−28.

[5] Application layer packet classifier for linux. 2009. http://l7-filter.sourceforge.net/

[6] Bro intrusion detection system. 2009. http://bro-ids.org/Overview.html

[7] TippingPoint X505. 2008. http://www.tippingpoint.com/products_ips.html

[8] Cisco IOS IPS deployment guide. http://www.cisco.com

[9] Yu F, Chen ZF, Diao YL, Lakshman TV, Katz RH. Fast and memory-efficient regular expression matching for deep packet inspection. In: Proc. of the IEEE/ACM Symp. on Architectures for Networking and Communications Systems. San Jose, 2006. 93−102. [doi: 10.1145/1185347.1185360]

[10] Kumar S, Dharmapurikar S, Yu F, Crowley P, Turner J. Algorithms to accelerate multiple regular expressions matching for deep packet inspection. In: Proc. of the ACM SIGCOMM. Pisa, 2006. 339−350. [doi: 10.1145/1159913.1159952]

[11] Becchi M, Cadambi S. Memory-Efficient regular expression search using state merging. In: Proc. of the IEEE Infocom. Anchorage, 2007. 1064−1072. [doi: 10.1109/INFCOM.2007.128]

[12] Kumar S, Chandrasekaran B, Turner JS, Varghese G. Curing regular expressions matching from insomnia, amnesia, and acalculia In: Proc. of the 3rd ACM IEEE Symp. on Architecture for Networking and Communications Systems. Washington: IEEE, 2007. 155−164.

[13] Becchi M, Crowley P. A hybrid finite automaton for practical deep packet inspection. In: Proc. of the ACM CoNEXT 2007. 2007. [doi: 10.1145/1364654.1364656]

[14] Smith R, Estan C, Jha S, Siahaan I. Fast signature matching using extended finite automaton (XFA). In: Proc. of the ICISS 2008. 2008. 158−172. [doi: 10.1007/978-3-540-89862-7_15]

[15] Hopcroft JE, Motwani R, Ullman JD. An Introduction Automata Theory, Languages and Computation. 2nd ed., Boston: Addison Wesley, 2000. 1−50.

[16] Chen SH, Su JS, Fan HP, Hou J. An FSM state table compressing method based on deep packet inspection. Journal of Computer Research and Development, 2008,42(8):1299−1306 (in Chinese with English abstract).

[17] Kong SJ, Smith R, Estan C. Efficient signature matching with multiple alphabet compression tables. In: Proc. of the 4th Int'l Conf. on Security and Privacy in Communication Networks. Istanbul, 2008. 1−10. [doi: 10.1145/1460877.1460879]

[18] Ficara D, Giordano S, Procissi G, Vitucci F, Antichi G, Pietro AD. An improved DFA for fast regular expression matching. ACM SIGCOMM Computer Communication Review, 2008,38(5):29−40. [doi: 10.1145/1452335.1452339]

[19] Kumar S, Turner J, Williams J. Advanced algorithms for fast and scalable deep packet inspection. In: Proc. of the IEEE/ACM Symp. on Architectures for Networking and Communications Systems. 2006. 81−92. [doi: 10.1145/1185347.1185359]

[20] Becchi M, Crowley P. An improved algorithm to accelerate regular expression evaluation. In: Proc. of the IEEE/ACM Symp. on Architectures for Networking and Communications Systems. 2007. 145−154. [doi: 10.1145/1323548.1323573]

[21] Zhang SZ, Luo H, Fang BX, Yun XC. Fast and memory-efficient regular expression matching using transition sharing. IEICE Trans. on Information and Systems, 2009,E92-D(10):1953−1960. [doi: 10.1587/transinf.E92.D.1953]

[22] Xu Q, E YP, Ge JG, Qian HL. Efficient regular expression compression algorithm for deep packet inspection. Journal of Software, 2009,20(8):2214−2226 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/3311.htm [doi: 10.3724/SP.J.1001.2009.03311]

[23] Myers G. A four russians algorithm for regular expression pattern matching. Journal of the ACM, 1992,39(2):432−448. [doi: 10.1145/128749.128755]

[24] Green T, Gupta A, Miklau G, Onizuka M, Suciu D. Processing XML streams with deterministic automata and stream indexes. ACM Trans. on Database Systems, 2004,29(4):752−788. [doi: 10.1145/1042046.1042051]

[25] Smith R, Estan C, Jha S, Kong SJ. Deflating the big bang: Fast and scalable deep packet inspection with extended finite automata. In: Proc. of the ACM SIGCOMM. 2008. 207−218. [doi: 10.1145/1402958.1402983]

[26] Clark CR, Schimmel DE. Efficient reconfignrable logic circuit for matching complex network intrusion detection patterns. In: Proc. of the 3rd Int'l Conf. on Field Programmable Logic and Application (FIL 2003). New York: Springer-Verlag, 2003. 956−959.

[27] Moscola J, Lockwood J, Loui RP, Pachos M. Implementation of a content-scanning module for an Internet firewall. In: Proc. of the 11th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Washington, 2003. 31−38. [doi: 10.1109/FPGA.2003.1227239]

[28] Hutchings BL, Franklin R, Carver D. Assisting network intrusion detection with reconfigurable hardware. In: Proc. of the 10th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Washington, 2002. 111−120. [doi: 10.1109/FPGA.2002.1106666]

[29] Sourdis I, Pnevmatikatos D. Pre-Decoded CAMs for efficient and high-speed NIDS pattern matching. In: Proc. of the 12th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Napa, 2004. 258−267. [doi: 10.1109/FCCM.2004.46]

[30] Becchi M, Crowley P. Efficient regular expression evaluation: Theory to practice. In: Proc. of the 4th ACM/IEEE Symp. on Architectures for Networking and Communications Systems. New York: ACM Press, 2008. 50−59. [doi: 10.1145/1477942.1477950]

[31] Lee J, Hwang SH, Park NS, Lee SW, Jun S, Kim YS. A high performance NIDS using FPGA-based regular expression matching. In: Proc. of the 2007 ACM Symp. on Applied Computing Archive. New York, 2007. 1187−1191. [doi: 10.1145/1244002.1244259]

[32] Lin CH, Huang CT, Jiang CP, Chang SC. Optimization of regular expression pattern matching circuits on FPGA. In: Proc. of the Conf. on Design, Automation and Test in Europe: Designers' Forum (DATE 2006). Leuven: European Design and Automation Association, 2006. 12−17. http://portal.acm.org/citation.cfm?id=1131355.1131359

[33] Faezipour M, Nourani M. Constraint repetition inspection for regular expression on FPGA. In: Proc. of the 2008 16th IEEE Symp. on High Performance Interconnects. Washington, 2008. 111−118. [doi: 10.1109/HOTI.2008.14]

[34] Sidhu R, Prasanna VK. Fast regular expression matching using FPGAs. In: Proc. of the 10th Annual IEEE Symp. on Field-Programmable Custom Computing Machines. Washington, 2001. 237−238. [doi: 10.1109/FCCM.2001.22]

[35] Yang YHE, Prasanna VK. Automatic construction of large-scale regular expression matching engines on FPGA. In: Proc. of the 2008 Int'l Conf. on Reconfigurable Computing and FPGAs. Cancun, 2008. 73−79. [doi: 10.1109/ReConFig.2008.47]

[36] Virtex-4 FPGA, Xilinx. 2006. http://www.xilinx.com

[37] Becchi M, Crowley P. Extending finite automata to efficiently match Perl-compatible regular expressions. In: Proc. of the Int'l Conf. on Emerging Networking Experiments and Technologies (CoNEXT). Madrid, 2008. [doi: 10.1145/1544012.1544037]

[38] Becchi M, Wiseman C, Crowley P. Evaluating regular expression matching engines on network and general purpose processors. In: Proc. of the 2009 ACM/IEEE Symp. on Architectures for Networking and Communications Systems. Princeton, 2009. [doi: 10.1145/1882486.1882495]

[39] Hayeng M, Shaw M. Extended finite automata on stream-oriented architectures using rapid mind. Technical Report, CS 758, Madison: University of Wisconsin-Madison, 2008. 1−11.

[40] Cao J, Tan JL, Liu P, Guo L. Research of Boolean expression matching. Application Research of Computers, 2007,24(9):70−72 (in Chinese with English abstract).

[41] Cao J, Liu YB, Liu P, Tan JL, Guo L. Research on ordered Boolean expression matching with window. Journal on Communications, 2007,28(12):125−130 (in Chinese with English abstract).

附中文参考文献:

[16] 陈曙晖,苏金树,范慧萍,侯婕.一种基于深度报文检测的 FSM 状态表压缩技术.计算机研究与发展,2008,42(8):1299−1306.

[22] 徐乾,鄂跃鹏,葛敬国,钱华林.深度包检测中一种高效的正则表达式压缩算法.软件学报,2009,20(8):2214−2226. http://www.jos.org.cn/1000-9825/3311.htm [doi: 10.3724/SP.J.1001.2009.03311]

[40] 曹京,谭建龙,刘萍,郭莉.布尔表达式匹配问题研究.计算机应用研究,2007,24(9):70−72.

[41] 曹京,刘燕兵,刘萍,谭建龙,郭莉.定序窗口布尔表达式匹配技术研究.通信学报,2007,28(12):125−130.

张树壮(1982－),男,河北卢龙人,博士生,主要研究领域为网络安全,信息内容安全.

方滨兴(1960－),男,博士,教授,博士生导师,CCF 高级会员,主要研究领域为信息安全,网络计算.

罗浩(1979－),男,博士,副教授,主要研究领域为网络安全.