一种正则表达式编译器优化技术

李朋飞,陈曙晖,徐成成
(国防科学技术大学 计算机学院,湖南 长沙 410073)

摘 要:正则表达式匹配在网络安全领域具有重要地位。传统的正则表达式匹配引擎通常采用 NFA 和 DFA,由于具有匹配性能高的特点,DFA 成为深度报文检测(DPI)的首选。但是 DFA 的生成首先需要由正则表达式转换成 NFA,再由 NFA 转换成 DFA,这个过程称为正则表达式的编译,且是一个计算非常密集的行为。针对构建 DFA 过程中耗时过多的问题,在 Michela Becchi 实现的编译过程的基础上,提出了一种基于多核平台的多线程并行化执行的方案,来降低构建 DFA 消耗的时间。同时针对所使用正则表达式中不能识别尾锚的不足,增加尾锚处理流程,提高正则表达式匹配的准确性。实验结果表明,经并行优化,构建 DFA 过程的加速比达到 2.3 及以上,且添加的尾锚处理流程经验证是正确的。

关键词:正则表达式;子集法;确定型有限自动机;并行;优化
中图分类号:TP393.08 **文献标识码**:A **文章编号**:1673-629X(2015)09-0123-06
doi:10.3969/j.issn.1673-629X.2015.09.027

A Regular Expression Compiler Improvement Technique

LI Peng-fei, CHEN Shu-hui, XU Cheng-cheng
(College of Computer, National University of Defense Technology, Changsha 410073, China)

Abstract: Regular expression matching plays an important role in network security domain. Traditionally, NFA and DFA could be used in regular expression matching. As DFA has the characteristic of high throughput, it becomes the preferred option of Deep Packet Inspection (DPI). However, DFA construction needs a compilation, which first converts regular expressions into NFA, and NFA is then used to construct DFA. The compilation process is a computation-intensive behavior. In this paper, the most time-consuming process of the construction of DFA is researched. Based on the previous works of Michela Becchi, propose a multi-thread parallel strategy to reduce time-cost in the compilation process on the multi-core platform. In addition, the function of tail anchor is added and the accuracy is proved, so that the regular expression matching engine can deal with tail anchor. The experimental results show that the compiling process can be accelerated by 2.3 times with parallel optimization.

Key words: regular expression; subset construction algorithm; DFA; parallel; optimization

0 引 言

深度报文检测技术(Deep Packet Inspection, DPI)是一种先进的流量分类与控制技术,广泛应用于网络安全领域。利用从流量中提取的精确字符串,该技术能够准确判断协议类型。然而,随着网络技术的飞速发展,各种新型协议层出不穷,新的攻击方式不断涌现,从网络流量中提取精确字符串特征的难度越来越大。因此急切需要一种表达能力强的语言来描述这些新型协议和恶意流量的特征。正则表达式[1]以其强大、灵活的表达能力,很好地填补了这一空缺。

目前,许多网络安全类应用使用正则表达式[2]进行流量分类,如 L7-filter[3], Snort[4] 以及 Bro[5] 等。L7-filter 是基于 Linux IPtable 的软件[6-7],其所有规则都采用正则表达式描述;Snort 中采用正则表达式描述的协议规则已达到三分之一;而 Bro 的协议规则全部采用正则表达式描述。

正则表达式是一种由普通字符和元字符组成的描述,用于说明一种模式,常用来将某个字符串(或报文体)与这个模式进行匹配[8]。通常,正则表达式匹配采用的是有限状态机[9-10](Finite State Machine, FSM)。FSM 分为两类:非确定型有限自动机(Nondeterministic Finite Automaton, NFA)和确定型有限自动

收稿日期:2014-10-28 **修回日期**:2015-01-29 **网络出版时间**:2015-08-26
基金项目:国家自然科学基金资助项目(61379148)
作者简介:李朋飞(1986-),男,硕士研究生,研究方向为网络安全、模式匹配;陈曙晖,副教授,博士,研究方向为计算机网络与网络安全。
网络出版地址:http://www.cnki.net/kcms/detail/61.1450.TP.20150826.1556.028.html

机(Deterministic Finite Automaton,DFA)。对于一个输入,DFA 只产生一种结果,而 NFA 则可能产生多种结果。相比于 NFA,DFA 不需要回溯,匹配性能更高,因此,DFA 成为 DPI 匹配引擎的首选[11]。

但是,构建 DFA 的过程存在密集计算,消耗大量时间,影响了 DFA 的匹配性能。针对这个问题,目前有两种解决途径:一种是硬件加速[12],另一种是软件加速[13-14]。相比前者,后者更易于实现,因而受到更多的关注。软件加速大致可分为两类,一类是提出新型的状态机或者改变正则表达式实现过程中的一些数据结构[15-17];另一类是基于多核平台对正则表达式进行并行化处理[18-19]。

另外,文中基于 Michela Becchi 实现的正则表达式编译过程展开算法优化工作[20]。在研究过程中发现该实现不能识别尾锚。尾锚是一种位置标记,如 QQ 规则以 03 结尾,它能够提高特征匹配的准确性。因此,为正则表达式添加尾锚功能是非常必要的。

针对以上提出的问题,文中主要研究工作如下:

(1)基于多核平台,对构建 DFA 过程进行多线程并行优化。对构建 DFA 的子集法进行深入研究,找到子集法编译过程的瓶颈,并利用多线程加以并行优化。

(2)添加尾锚功能。为所用正则表达式添加尾锚识别功能,提高正则表达式的准确性。

1 子集法

子集法是将 NFA 确定化为 DFA 的经典算法,在生成 DFA 过程中起关键作用。将正则表达式编译为 DFA 的过程如图 1 所示。

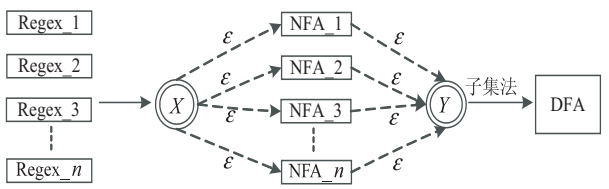

图 1 多条正则表达式编译为一个 DFA 的过程

如图 1 所示,先将 n 条正则表达式编译为一个 NFA,然后通过子集法把 NFA 转换为 DFA。其中,Regex_ n 表示第 n 条正则表达式;NFA_ n 表示由第 n 条正则表达式编译的 NFA; X 表示整个 NFA 的起始状态; Y 表示整个 NFA 的结束状态。由各条正则表达式编译的 NFA 通过空转换 ε 与 X 和 Y 相连。当所有正则表达式都被编译为 NFA 之后,运用经典子集法将 NFA 转换为 DFA。

经典子集法的伪代码如图 2 所示。

图中,DFA 五元组是 $D=(Q,F,\Sigma,\delta,q_0)$。其中, Q 为 DFA 状态集合; F 为终止状态集合; Σ 为字母表; δ 为跳转函数; q_0 为初始状态。NFA 五元组是 $N=(Q',F',\Sigma',\delta',q_0')$,各参数含义同 DFA。

算法 1: 经典子集法

1: producer NFA2DFA($N=(Q',F',\Sigma',\delta',q_0'),D=(Q,F,\Sigma,\delta,q_0)$)
2: $I\leftarrow\{\varepsilon(q_0')\}$
3: push(queue, I)
4: while queue is not empty do
5: $S\leftarrow$pop(queue)
6: $Q\leftarrow Q\cup\{S\}$
7: if $S\cap F'\neq\varnothing$ then
8: $F\leftarrow F\cup\{S\}$
9: end if
10: processed←processed $\cup\{S\}$
11: for each $c\in\Sigma$ do
12: $T\leftarrow\bigcup_{s\in S}\delta'(s,c)$
13: $R\leftarrow$check(processed, T)
14: if R =FALSE then
15: push(queue, T)
16: $\delta(S,c)\leftarrow T$
17: end if
18: end for
19: end while
20: end producer

图 2 经典子集法算法描述

利用子集法构造 DFA 是从获得 NFA 初始状态 q_0' 的 ε 闭包 $\varepsilon(q_0')$ 开始的。首先以 $\varepsilon(q_0')$ 为初始子集合,按照跳转函数 $|\delta'(s,c)|$ 查找出 $\varepsilon(q_0')$ 中每个状态在输入字符 c 时跳转得到的下一个状态,合并成新的状态子集合 $T\leftarrow\bigcup_{s\in S}\delta'(s,c)$,对应第 12 行。接着判断新子集合是否已被经处理过,此操作对应第 13 行。若产生的子集合未被处理(R =FALSE),则该子集合是新产生的,将其加入到待处理队列,对应第 15 行;否则只作一般处理。每次得到新的子集合,就将其映射为 DFA 状态表的一个状态,子集合成员的可接受状态由该 DFA 状态接受。重复以上操作,直到没有新的子集合出现为止,子集法构造 DFA 过程才全部结束。

如图 2 所示,在经典子集法执行过程中,第 12 行操作耗费时间最多。最坏情况下,集合 $|S'|$ 和跳转集合 $|\delta'(s,c)|$ 的大小与 $|Q'|$ 相同,该操作的时间复杂度为 $O(|S'||\delta'(s,c)|)=O|Q'|^2$。最坏情况下,整个子集法的时间复杂度为 $O(|Q||\Sigma||S'||\delta'(s,c)|)=O(|Q||\Sigma||Q'|^2)$。

本节设计一组实验,测试并记录正则表达式编译过程中构造子集合与子集法整体分别花费的时间。所用规则来自基于 L7-filter 进一步完善的特征集,如表 1 所示。

该实验采用的计时方法是:在正则表达式编译过程中运行计时器,分别对子集合构造过程和 NFA 转 DFA 过程进行计时。为体现普遍性,在该实验中,将 DFA 状态数控制在一万以上;同时,为了控制实验花费时间在可接受范围内,最多选取 13 条规则。最终,选择四组规则数目不同的规则集进行实验,实验结果如表 2 所示。

表 1　实验所用规则

序号	正则表达式
1	^. ?. ? \x02. +\x03 $
2	^220[\x09-\x0d -~] * ftp
3	^(ver [-~] * msnftp\x0d\x0aver msnftp\x0d\x0ausr|method msnmsgr:)
4	ver [0-9]+ msnp[1-9][0-9]? [\x09-\x0d -~] * cvr0\x0d\x0a $ |usr 1 [! -~]+ [0-9.]+\x0d\x0a $ |ans 1 [! -~]+ [0-9.]+\x0d\x0a $
5	^(ymsg|ypns|yhoo). ?. ?. ?. ?. ?. ?. ? [lwt]. * \xc0\x80
6	fetion(\. com\. cn)? |Host: . * feixin\. 10086\. cn\x0d\x0a
7	(GET |Host:)[^\x0d\x0a] * mop\. com\x0d\x0a
8	(GET |Host:)[^\x0d\x0a] * (renren\. com|xiaonei\. com) \x0d\x0a
9	Host: [^\x0d\x0a] * (weibo\. com|weibo. cn|api. weibo. cn\t. sina. cn) \x0d\x0a
10	Host: [^\x0d\x0a] * sinaimg\. cn\x0d\x0a. * User-Agent. * weibo[^\x0d\x0a] * \x0d\x0a
11	Host: [^\x0d\x0a] * (tcli\. 3g\. qq\. com|mbloghead\. store\. qq\. com|t\. qq\. com|t. * \. qlogo\. cn|t. * \. qpic\. cn) \x0d\x0a
12	^. . (\x01\x00|\x81\x80)
13	^220[\x09-\x0d -~] * (e? smtp|simple mail)

表 2　构造子集合花费时间与 NFA 转 DFA 花费时间统计表

规则数目	NFA 状态数	转换时间/s	DFA 状态数	构造时间/s	构造时间占转换时间的比例/%
9	226	141	18 165	129	91.5
10	55	610	56 490	555	91.0
11	316	1 473	124 211	1 321	89.7
13	337	1 565	127 067	1 392	88.9

其中,规则数目指从表 1 中按序号顺序取出的规则数目;NFA 状态数指待转换的 NFA 状态数;转换时间指 NFA 转 DFA 花费的时间;DFA 状态数指子集法构造的 DFA 状态数目;构造时间指在 NFA 转 DFA 过程中,构造所有子集合花费的时间;构造时间占转换时间的比例表示构造所有子集合的时间除以子集法整体花费时间得到的百分比。

实验环境如下:

(1)操作系统:Linux 64;

(2)CPU:Inter core i3-2130 四核,主频 3.40 GHz;

(3)内存:4G。

文中后续的其他实验,其实验环境均与本次实验相同。

从表 2 的实验数据可以看出,随着规则数目的增加,NFA 和 DFA 的状态数不断增多,且 DFA 状态数目总是比 NFA 状态数目多出许多。这是子集法转换过程中必然出现的情况。

另外,从实验数据中发现,子集法所用时间和构造子集合所用时间都随着规则数目增多而增加;规则数目增加幅度越大,子集法和构造子集合所用时间增加越快。

分析表 2 列出的构造时间占转换时间的比例发现,构造子集合时间所占比例随着规则数目的增加而减少。文中把 NFA 转 DFA 分为构造子集合与子集合查重两个子过程,每个新构造的子集合都要经过子集合查重。随着 NFA 转 DFA 过程中构造的子集合数目的增加,子集合查重被频繁调用,这两个子过程花费时间都在增大,而子集合查重时间增加速度大于构造子集合时间,导致构造子集合花费时间在整个子集法中占用时间比例下降。

总体看来,在选取不同数量的规则进行实验时,构建子集合花费时间所占比例都达到 88% 以上。

据此,可以得出结论:子集法的瓶颈在于构造子集合的过程。

2　利用 OpenMP 并行化处理构造子集合过程

OpenMP[21] 是基于线程的并行编程模型,由一组编译制导指令、运行时库函数(Run-Time routines)和环境变量组成,能够使用户在保证程序的可移植性的前提下,按照标准将已有的串行程序逐步并行化。OpenMP 具有操作简单、移植性好和可扩展等优点。

OpenMP 采用 Fork-Join 并行执行模型,在并行模式与串行模式之间不断转换。

Fork-Join 模型的执行过程如下:

(1)Fork:主线程创建一个并行线程队列,然后,并行域中的代码在不同的线程队列中并行执行;

(2)Join:当并行域执行完之后,它们或被同步或被中断,最终只有主线程仍然执行。该模型十分灵活,可根据具体需要自由使用,且可以被重复调用。

使用 OpenMP 应注意的问题:

(1)确定可并行处理代码块的信息;

(2)待处理的代码块必须是单入口单出口;

(3)控制好并行域中各个参数的属性(私有或共享);

(4)该并行域中各参数之间无相关性,即前一循环产生的数据对后续的循环无影响。

2.1 并行化处理构造子集合过程

经研究,构造子集合的过程是一个循环,且该循环的特点符合 OpenMP 并行化的要求,能够实现并行化处理。

在并行化处理子集法实验中,遇到了数据相关性问题:每产生一个新的子集合,都需要检查该子集合的重复性,而此操作必须遍历所有已处理的子集合。针对该问题,本实验拟采用两种解决方法:

第一种方法实行模块化管理,把并行化处理子集合构造模块产生的数据输入到子集合查重模块,由这两个模块共同完成 NFA 转 DFA 过程;

第二种方法采用 Fork-Join 模型,当程序运行到检查子集合重复性的步骤时,暂停并行模式(Fork)的执行,切换到串行模式(Join),以避免具有相关性的参数混杂到并行域中。

针对以上两种方法,设计实验比较两种方法对子集法的优化效果。

该实验采用表 1 中列出的正则表达式,每次选取不同数目的规则,分别采用上述两种方法处理数据相关性问题,分别统计子集法执行时间,得出的结果如表 3 所示。

其中,规则数目指从表 1 按序号依次取出的规则数目;经典子集法指经典子集法花费的时间;方法一和方法二分别指在两种方法下子集法花费的时间;方法二加速比指经典子集法花费的时间除以经方法二优化的子集法所花费时间得到的值,该数据将作为子集法优化的重要参数加以详细分析。

从表 3 的实验数据很容易看出,针对子集法的优化,在不同规则数目下,采用方法二执行子集法的时间普遍小于采用方法一的。这充分说明,在对子集法进行优化时,方法二优于方法一。

表 3 两种方法的优化效果

规则数目	经典子集法/s	方法一/s	方法二/s	方法二加速比
9	141	72	62	2.3
10	610	308	260	2.3
11	1 473	765	640	2.3
13	1 565	920	663	2.4

方法一不如方法二的原因是:方法一采用模块化管理,增加了子集合的存取次数,而方法二直接处理子集合,省去存取子集合的操作。具体来讲,方法一必须先将子集合存储到内存中,待所有子集合全部产生之后再对子集合进行统一查重。查重操作访问已经存储的子集合,增加了访存次数,导致耗费时间增多。而方法二则不需要存取子集合,其时间消耗较方法一大大减少。

综上分析,最终选用方法二来优化子集法,具体的优化步骤及实验将在下节展开。

2.2 优化结果及分析

根据 2.1 节的实验结论,对并行过程中遇到的数据相关性问题采用并行转串行的方法加以解决,即采用 Fork-Join 模型对子集法进行并行化处理。针对表 2 的统计结果,利用上述方法对相应的规则集进行并行化处理,得到的实验结果如表 3 所示。在表 3 中,方法二花费的时间即为经优化的子集法所花费时间,方法二加速比即为优化得到的加速比。

如表 3 所示,实验中,在规则数目不同的情况下分别进行优化实验,并计算子集法优化加速比,加速比均达到 2.3 及以上。实验使用四个线程进行加速,加速比达到 2.3 及以上,说明并行优化加速达到明显效果。

比较并行优化前后子集法的执行时间,发现经过并行优化,子集法执行时间大幅减少;但是,优化加速比未达到 4。造成这种结果的原因是:并行执行过程中,OpenMP 按照字母表(如 ASCII)的大小自动将任务平均分配给四个线程,但是构造子集合的过程中,NFA 状态并非经每个字符都发生跳转,这样造成各个线程实际执行的任务量不等,未能真正把任务平均分配到各个线程。各线程任务量不等,使得各线程运行时间不等,运行时间短的线程要等待运行时间长的线程执行完毕才能继续下一步骤。该并行过程消耗的时间取决于运行时间最长的线程。最终,构造子集合过程中各线程分配任务不等造成子集法优化加速比未达到预期加速比 4。

另外,在表 3 中,当规则数目增加到 13 时,优化加速比达到 2.4。出现这一结果的原因是:在该实验中,相比其他规则集,OpenMP 将任务更均匀地分配到各个线程,达到更优的加速效果。

3　尾锚添加

在正则表达式中,尾锚作为一种位置标记,通常有两种解释:一种将尾锚定义为字符串的结束位置;另一种将尾锚定义为行的结束位置。文中采用第一种解释。

文中所使用的正则表达式不具有尾锚识别功能,但是在实际应用中尾锚是必不可少的。例如,表 1 中,规则 1 需要将字符\x03 定位到字符串结束位置,规则 4 需要将字符\x0a 定位到字符串结束位置等。匹配过程中将这些字符准确定位到字符串结尾,以达到提高特征匹配准确性的目的。因此,必须给所用正则表达式添加尾锚。

3.1　可行性

如第一节所述,在子集法构造 DFA 过程中,原有子集合通过跳转函数产生新子集合。若原有子集合中某个状态经跳转函数跳转到终止状态,那么该终止状态必被包含在新子集合中,同时终止状态包含的可接受规则 ID 也将被新子集合接受;而一个子集合最终被映射为 DFA 的一个状态,子集合包含的可接受规则 ID 被相应 DFA 接受。可以得出结论:终止状态的可接受规则 ID 能够被 DFA 继承。因此,将尾锚与可接受规则 ID 关联起来,一定能将尾锚正确地传递给 DFA。

3.2　方案与验证

本实验设计两种方案添加尾锚功能。分别是:将尾锚作为一个特殊对象,添加到 NFA 终止状态;在处理终止状态时,将尾锚作为一个标记,建立尾锚标记与可接受规则 ID 的一一对应关系。

两种方案的具体实验过程与验证分析如下:

(1)采用第一种方案添加尾锚标记,分别进行两组实验。

第一组实验,只编译拥有尾锚的正则表达式(如 QQ 规则),并输入相应的字符串(如 QQ 报文),观察匹配结果。结果表明,正则表达式匹配成功,尾锚正确匹配。

第二组实验,把含尾锚的正则表达式和不含尾锚的正则表达式混在一起,编译为一个 DFA。这时要考虑两条正则表达式的结束字符是否相同,对这两种情况分别进行实验。实验中使用的正则表达式是随机选取的。首先编译两条结束字符不同的正则表达式 abcde 和 hijk $,输入含有相应正则表达式的字符串,观察匹配结果。输入的字符串都发生匹配。然后编译两条结束字符相同的正则表达式 abcde 和 gfdce $,输入含有相应正则表达式的字符串,观察匹配结果。结果表明,每次只能匹配含尾锚的正则表达式,而不能匹配不含尾锚的正则表达式。

综上,该方案添加的尾锚标记干扰不含尾锚的正则表达式的匹配,因此舍弃该方案。

(2)采用第二种方案添加尾锚标记。该方案针对第一种方案容易混淆尾锚标记的问题,采取建立规则 ID 与尾锚标记之间一一对应关系的方法来进行改进。在该方法中,当一个状态为终止状态时,该状态自动存储相应的可接受规则 ID,在存储可接受规则 ID 的同时添加尾锚标记,建立尾锚标记与规则 ID 之间的一一对应关系。

该方案的具体原理如下:以 QQ 规则(^.?.? \x02.+\x03 $)为例。假设 QQ 在规则集中的 ID 是 1,则 QQ 规则 ID 与尾锚标记对应关系如图 3(a)所示。

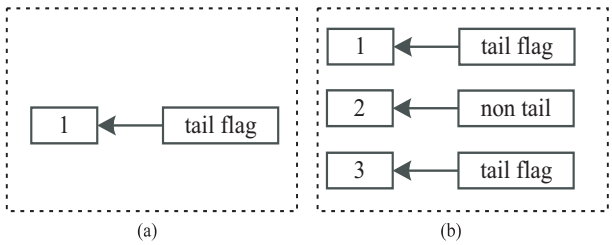

图 3　规则 ID 与尾锚标记之间的对应关系

由尾锚标记与规则 ID 的一一对应关系可知,当自动机跳转到终止状态时,它能通过规则 ID 获知该规则是否需要匹配尾锚。

规则匹配最复杂的情形是:在同一个可接受状态匹配多条规则。可以证明,依据这种方案,即使在最复杂情形下,尾锚依然能够匹配成功。

假设一个规则集中包含以下正则表达式:(1)abcdef $;(2)bcdef;(3)cdf $。在经字符 f 跳转到终止状态时会出现三条规则同时被接受的情况,且并非都具有尾锚。该规则集经编译之后,规则 ID 与尾锚标记之间的对应关系如图 3(b)所示。

从图中可以看出,由于规则 ID 与尾锚标记之间是一一对应关系,匹配过程中,自动机不会将无尾锚的规则 2 误认为拥有尾锚,规则 1 与规则 3 包含的尾锚都能被正确识别。

根据上述原理,设计两组实验验证方案二的正确性。第一组实验,只编译 QQ 的正则表达式,然后输入 QQ 报文,观察匹配结果。实验结果表明,QQ 规则匹配成功,尾锚匹配正确。

第二组实验,同时编译的正则表达式:(1)abcdef $;(2)bcdef;(3)cdf $。然后依次输入相应的字符串进行匹配,观察匹配结果。所有正则表达式都能匹配成功,添加的尾锚标记不干扰不含尾锚的正则表达式的匹配。

综合以上两组实验的结果,可以得出结论:方案二添加的尾锚标记不会对无尾锚的规则产生影响,该方案添加尾锚成功。但是,也要注意到,添加尾锚标记之后,自动机的每个状态都增加了尾锚标记,这将导致正

则表达式编译过程的空间开销增大。且每次匹配终止状态都需要查看是否存在尾锚标记,增加了额外操作。总的来说,添加尾锚功能提高了正则表达式匹配的准确性。

4 结束语

文中针对 Michela Becchi 实现的正则表达式编译过程存在的问题加以改进,利用多核平台对子集法进行并行加速。实验结果表明,优化加速比达到 2.3 及以上,使正则表达式的整体性能得到大幅提升;另外,针对该正则表达式无法识别尾锚的问题添加尾锚处理流程,经实验验证,该流程是有效的,提高了正则表达式匹配的准确性,能够有效解决协议识别等应用中定位报文尾部模式的需求。

对于正则表达式子集法的优化还有许多方面可以挖掘,接下来,将继续深入研究子集法构造过程的优化问题,从更多层面优化正则表达式,提高其整体性能。

参考文献:

[1] 陈火旺,刘春林,谭庆平,等. 程序设计语言编译原理[M]. 北京:国防工业出版社,2003.

[2] 张树壮,罗 浩,方滨兴. 面向网络安全的正则表达式匹配技术[J]. 软件学报,2011,22(8):1838-1854.

[3] L7-filter[EB/OL]. 2014-03-20. http://l7-filter.clearfoundation.com/.

[4] Snort user manual, the snort project[EB/OL]. 2014-03-20. https://www.snort.org/documents/snort-users-manual/snort_manual.pdf.

[5] Bro manual[EB/OL]. 2014-05-01. https://www.bro.org/sphinx/index.html.

[6] Yu Fang, Chen Zhifeng, Diao Yanlei. Fast and memory-efficient regular expression matching for deep packet inspection[R]. Berkeley: University of California, Berkeley, 2006.

[7] 范慧萍,宣 蕾,陈曙晖,等. 基于正则表达式的应用层协议识别加速[J]. 计算机研究与发展,2008,45(S):438-443.

[8] Jeffrey E F. 精通正则表达式[M]. 余 晟,译. 第 3 版. 北京:电子工业出版社,2012.

[9] Hopcrof J, Ullman F. Introduction to automata theory, languages, and computation[M]. 3rd ed. Boston: Addison Wesley, 2007.

[10] 王 柏,杨 娟. 形式语言与自动机[M]. 北京:北京邮电大学出版社,2003.

[11] 李 璋,杜慧敏,张丽果. 基于分布式存储的正则表达式匹配算法设计与实现[J]. 计算机科学,2013,40(3):74-76.

[12] 陈曙晖,徐成成. 基于两级存储的正则表达式匹配技术[J]. 通信学报,2014,35(6):47-55.

[13] 李鲲鹏,兰巨龙,李玉峰. 基于 DFA 结构的高速并行正则表达式匹配算法[J]. 小型微型计算机系统,2013,34(5):1050-1053.

[14] 贺 炜,郭云飞,莫 涵,等. 基于多字符 DFA 的高速正则表达式匹配算法[J]. 计算机应用,2013,33(8):2370-2374.

[15] Gong Y, Liu Q, Shao X, et al. A novel regular expression matching algorithm based on multi-dimensional finite automata[C]//Proc of 15th international conference on high performance switching and routing. [s.l.]: IEEE, 2014: 90-97.

[16] Liu Yanbing, Guo Li, Guo Muyi, et al. Accelerating DFA construction by hierarchical merging[C]//Proc of ninth IEEE international symposium on parallel and distributed processing with applications. [s.l.]: IEEE, 2011.

[17] Yu Xiaodong. Deep packet inspection on large datasets algorithmic and parallelization techniques for accelerating regular expression matching on manycore[D]. Missouri: University of Missouri, 2013.

[18] Lin Cheng-Hung, Liu Chen-Hsiung, Chang Shih-Chieh. Accelerating regular expression matching using hierarchical parallel machines on GPU[C]//Proc of IEEE Globecom 2011. [s.l.]: IEEE, 2011: 1-5.

[19] Becchi M, Crowley P. Efficient regular expression evaluation: theory to practice[C]//Proceedings of the ACM/IEEE symposium on architectures for networking and communications systems. [s.l.]: IEEE, 2008: 50-59.

[20] Becchi M. Data structures, algorithms and architectures for efficient regular expression evaluation[D]. Washington: Washington University, 2009.

[21] 陈国良. 并行计算—结构·算法·编程[M]. 北京:高等教育出版社,2003.

一种正则表达式编译器优化技术

作者： 李朋飞， 陈曙晖， 徐成成， LI Peng-fei， CHEN Shu-hui， XU Cheng-cheng
作者单位： 国防科学技术大学 计算机学院,湖南 长沙,410073
刊名： 计算机技术与发展 ISTIC
英文刊名： Computer Technology and Development
年，卷(期)： 2015(9)