高技术通讯 2009 年 第 19 卷 第 6 期:551 ~ 557

doi:10.3772/j.issn.1002-0470.2009.06.001

一种面向大规模特征集的高效特征匹配算法①

张 伟②* 薛一波** 周宗伟* 汪东升***

(*清华大学计算机系 北京 100084)

(**清华大学网络安全研究中心 北京 100084)

(***清华大学微处理器与片上系统研究中心 北京 100084)

摘 要 针对传统特征匹配(网络和信息安全系统的核心技术)算法的性能随着特征集规模的不断增大而不断下降的问题,提出了一种面向大规模特征集的高效特征匹配算法 ALPM。该算法基于传统算法 WM 的跳跃思想,并结合硬件体系结构的特点,对预处理过程和匹配过程分别采用了不同的优化策略,如采用不同的哈希函数索引 Shift 表和 Hash 表,在预处理过程中动态截取特征标志,在匹配过程中结合 Cache 大小和特征集规模调整哈希函数冲突概率等,以提高匹配的性能。实验结果表明,针对大规模特征集,ALPM 算法匹配性能比经典算法提高 5 ~ 10 倍。

关键词 大规模特征集, 特征匹配, 字符串匹配, 哈希冲突, 多线程技术

0 引 言

全球计算机网络的飞速发展,给整个社会的经济、科技、文化带来了巨大的推动和冲击。网络传播着大量有用的信息,同时,网络上也存在大量的不良信息,例如计算机病毒、网络攻击、垃圾邮件、色情信息、反动言论等,这些不良信息不仅会造成重大的经济损失,而且会干扰国家的政治、科技、国防、宗教等的正常秩序,干扰人民群众的正常生活,甚至引发社会动荡。因此,如何才能及时、准确、安全地在庞大的实时网络信息流中监测、识别特定信息,以及如何抵制、消除这些信息,已经成为网络和信息安全领域的关键问题和热点问题。

特征匹配(pattern matching)是网络和信息安全应用中广泛使用的关键技术,比如防火墙、病毒检测、入侵检测与防御、内容过滤、网络审计等系统核心处理流程都采用了这种技术。但随着网络的发展,特征集(pattern set)规模在不断扩大,比如开源网络入侵检测系统 Snort[1] 从最开始的几百条变为现在的几千条,开源防病毒系统 ClamAV[2] 目前的病毒定义已经超过 20 万条,并且每天都在增加,与此同时,基于网络内容安全的应用也越来越多,比如统一资源定位符(URL)过滤系统、敏感信息过滤系统、网络审计系统,它们的特征集小则几百条,多则上万条,甚至几十万条。由于传统的特征匹配算法在特征集增大的情况下性能不断下降[3,4],因此本文在分析传统特征匹配算法和系统的基础上,提出了适用于大规模特征集的改进型算法 ALPM(architectural large-scale pattern matching, ALPM)。实验结果表明,ALPM 算法的特征匹配性能与经典算法相比有大幅度的提高。

1 相关工作

当前多特征精确匹配技术的主要研究思路可以大致分为两类:一类是研究软件特征匹配算法的改进以提高性能,一类是采用专用硬件(如现场可编程门阵列(FPGA)、网络处理器、专用集成电路(ASIC)等)来提高匹配性能的技术,应该说,基于专用硬件的算法和结构研究是目前学术界的一个热点,但硬件开发成本较高,开发周期较长,且更新不灵活,特征集规模也受限于芯片的门数量,因此,目前工业界基本上都是采用软件特征匹配算法或其改进算法。

软件特征匹配算法按照设计思路可以分为三类:(1)以 AC[5]、AC-BM[6] 等为代表的状态机算法;(2)以 BM[7]、WM[8] 等为代表的跳跃式算法;(3)以 SBDM[9]、SBOM[10] 等为代表的基于子串搜索的算

① 863 计划(2007AA01Z468)资助项目。

② 男,1980 年生,博士生;研究方向:计算机系统结构,网络与信息安全;联系人,E-mail: zhwei02@mails.tsinghua.edu.cn

(收稿日期:2008-05-06)

法。一般来说,状态机算法具有较好的最坏性能,影响算法性能的因素较少,但是需要的存储空间随着特征规模的增大而急剧膨胀,面对大规模特征集,访存成为状态机类算法的性能瓶颈。跳跃式算法一般具有较好的平均性能,但影响算法性能的因素较多,一般采取哈希(Hash)过滤的方法。随着特征集规模的增大,过滤效果越来越差,跳跃概率越来越小,面对大规模特征集,WM 等跳跃算法几乎无法跳跃,性能急剧下降。基于子串搜索的算法实现比较复杂,在预处理阶段构建自动机过程与 AC 很相似,在规模小时性能较好,但随着特征集的增大,消耗的存储空间依然会急剧膨胀,同时预处理时间也大大延长,无法满足大规模特征匹配的要求。

综上所述,针对多特征精确匹配问题,产生了上述经典算法以及很多它们的改进型算法[11-16],但随着实际应用特征集规模不断扩大,这些算法性能不断下降,如图 1(算法参照上述参考文献由作者实现,测试平台见 3.1 节)所示。而基于硬件的结构往往只支持小规模特征集,面对大规模特征集,往往需要大量片外存储,由于访存问题会造成性能的极大损失。从以上情况看,本文研究能适应于大规模特征集的特征匹配算法。

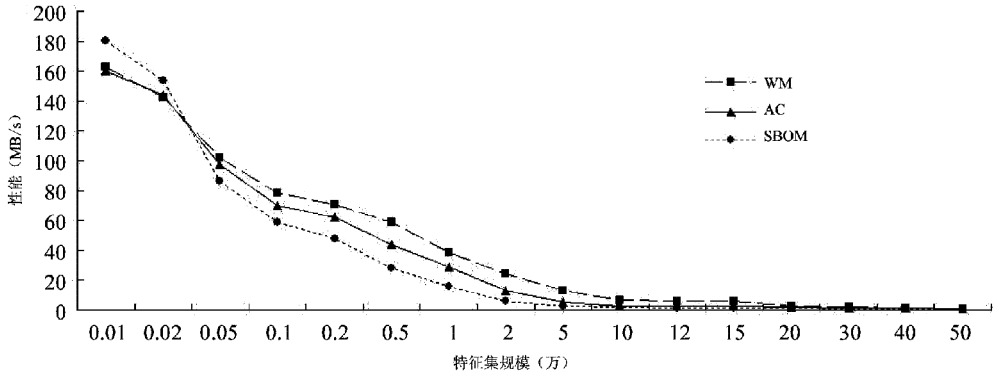

图 1 传统特征匹配算法在特征集规模变化下的性能下降趋势

2 支持大规模特征集的 ALPM 算法

2.1 WM 算法

台湾国立中正大学的 Wu 和美国亚利桑那大学的 Manber 于 1994 年提出了 WM 算法[8],其基本思想是将 BM 的启发式跳跃策略用于多特征匹配。预处理时,通过处理所有特征来建立跳跃表 Shift、哈希表 Hash 以及前缀表 Prefix,扫描时,首先计算文本当前被扫描的 B 个字符的哈希值,作为索引查找 Shift 表,取出跳跃值,若不为零则移动输入文本,若为零则通过 Hash 表取出将此 B 个字符作为后缀的所有特征,并查 Prefix 表取出前缀,先判断前缀与输入文本前缀是否匹配,若匹配,再进一步判断该条特征是否匹配。

从 WM 算法的原理上看,其性能的提升源于跳跃的发生,也就是在某些情况下不必每个字符都检验,基于初始化阶段建立的 Shift 表可以避开无谓的比较操作,从而可提升算法的性能。WM 算法每次读入 2 字节,则 Shift 表项为 2^{16},约为 6 万多项,在大规模特征集时,绝大部分表项为 0,表示大部分情况下无法跳跃,算法性能迅速下降。若特征集规模达到几十万规模时,几乎无法跳跃,并且每一步都需要额外匹配对应 Hash 表项所链接的多条特征,性能无法满足实际需求。

2.2 支持大规模特征集的 ALPM 算法

本文对 WM 算法实现的各个步骤进行了详细的分析,并结合硬件体系结构特点,对预处理过程和匹配过程分别提出不同的优化策略和技术:

(1) 4 字节索引建表

基于跳跃的算法都是尽可能利用跳跃来避免不必要的匹配操作,一旦无法跳跃,则进行逐个字节匹配,性能立刻下降。传统 WM 算法大都使用 2 字节提取特征属性,在特征数量较少(0.1K ~ 5K),并远小于 65536(2^{16})时有效,特征数量越大,Shift 表中 0 值越多,一旦接近或超过上限 65536,则扫描时基本不能跳跃,此时算法效率极低。为了能够支持大规模特征集,本文采用 4 字节字符的哈希值作为 Shift 表和 Hash 表入口,这样做的好处是:第一,支持百万规模特征集,增加跳跃概率;第二,中文以及亚洲字符大都采用双字节编码,即两个字节表示一个字,采用 4 字节可以更加方便有效地处理此类字符;第三,目前处理器芯片位宽大都为 32 位,可以有效地同时处理 4 字节数据。

(2) 双哈希索引策略

传统 WM 算法在对 Shift 表和 Hash 表进行访问时,地址采用的是 2 字节字符输入数据,而采用 4 字节进行索引时,为了提高空间存储效率,不可能采用 4 字节全地址,因为这样需要 2^{4*8}共计 4G 的存储空间,本文从 4 字节 32 位数据中分别提取 N1 位和 N2 位数据作为 Shift 表和 Hash 表入口地址,这样节省了大量空间,但同时也带来了哈希冲突的问题,即多个不同的 4 字节映射到 Shift 表和 Hash 表的同一地址上。一般来说,N1 和 N2 位数越多,占用空间越大,冲突越小,反之,位数越少,占用空间越小,冲突越大。实际测试表明,N1 和 N2 需要根据特征规模数来调整,一般保持 $PatternNum/2^{N1} < 3\%$ 时算法性能较好。为了减少哈希冲突,本文中,Shift 表与 Hash 表采用两个不同的哈希函数,尽可能保留输入字符数据的有效信息,充分利用两次哈希过程,覆盖全部信息,减少误查找概率,同时在编程时利用 inline 技术减少函数调用的开销。

(3) 预处理流程优化

WM 算法的预处理过程是基于给定特征集合建立 Shift 表和 Hash 表。假设特征集中最短特征长度为 m,每周期读取的字符块包含 B 个字符,m 大于 B,Shift 表表示跳跃值,初始化值都为 $m - B + 1$,表示最大的跳跃距离。Hash 表链接的是需要进一步匹配的特征地址,初始化所有链接为空。针对特征集中的每一条特征,从末尾往前截取 B 个字符,并以此 B 个字符的哈希值($B = 2$ 为本身)作为访问 Shift 表和 Hash 表的地址,更新对应 Shift 表项为 0,并将此特征链接到 Hash 表对应项里,然后往左偏移 1 个字符从特征串中再取 B 个字符,更新 Shift 表项为 1 和原表项值中的小值,依次往左偏移 1 个字符,更新 Shift 表项值,直到特征串开头。按照上述过程处理特征集中的所有特征,完成预处理流程。

由此不难发现,Shift 表控制跳跃距离,为了有更多的跳跃机会,在随机概率意义下,Shift 表为 0 的项越少越好。Hash 表链接的是需要进一步匹配的特征,当访问 Hash 表时需要判断其所链接的所有特征是否匹配,因此链接的特征越少越好。本文取 4 字节作为数据输入,Shift 表和 Hash 表地址位数分别为 2 个哈希函数的结果输出 N1 位和 N2 位,由于两次哈希函数不同,可以通过控制 Shift 表和 Hash 表的构造过程,使得 Shift 表为 0 的项更集中,同时使 Hash 表链接的特征数变少。下面介绍本文提出的优化预处理过程。

传统 WM 算法将特征的最后 B 个字符作为 Shift 表和 Hash 表的入口地址。由于本文采用不同的哈希函数访问 Shift 表和 Hash 表,因此,通过截取特征不同位置的 B 个字符可以使得同一特征有着不同的入口地址,从而改变特征集合在 Shift 表和 Hash 表的分布。本文给每个特征增加一个 offset 变量,标志特征串的 B 个字符在特征串中的位置,具体指的是 B 个字符中最右一个距离特征右端的距离。预处理过程中,在每个 offset 处读取 B 个字符,通过哈希函数计算出 Shift 表和 Hash 表的入口地址,选出符合条件的最右的 offset 并保存,使得特征集在 Shift 表中的分布更加集中,同时在 Hash 表中的分布更加广泛。然后按照传统方法从偏移值 offset + 1 处开始读取大小为 B 的字符块,计算出 Shift 表入口地址,更新为 1 和原表项值中的小值,依次往左偏移 1 个字符,更新 Shift 表项值,直到字符串开头。因为字符串长度越长,可以选择的空间越大,因此先处理短的特征,然后处理长的,完成 Shift 表和 Hash 表的更新。确定 offset 的伪代码如图 2 所示。

```
for(i=min_length;i++;i<=max_length)
  for(j=0;j++;j<PatternNum)
    If(Pattern[j].length == i){
      Counter = 0;
      for(k=0;k<i-min_length;kt++){
        Suffix = Pattern[k+min_length-4,k+min_length-1];
        If(Shift[Hash1(suffix)] == 0)
          Flag_Shift[k] = 1;
        If(Hash[Hash2(suffix)].linkpatternnum == 0)
          Flag_Hash[k] = 1;
        If( (Flag_Shift[k]) && (Flag_Hash[k]) )
          Counter++;
      }
      If (Counter ==1){
        Find k where (Flag_Shift[k]) && (Flag_Hash[k]) )
        Pattern[j].offset = k;
      }
    }
```

图 2　改进后的优化预处理过程伪码

本文提出的优化预处理过程的实施要付出一定的代价:一是增加了预处理时间,而由于预处理只需要运行一次,所以影响不大;二是每一条特征增加一个整型存储变量保存偏移值;三是在匹配过程中,传统方法是截取的最后 B 个字符,精确匹配需要从输入文本往前偏移(pattern _ length-B)开始,到当前位置结束,优化方法中,精确匹配从输入文本往前偏移(pattern _ length-B-offset)开始,到往后偏移 offset 结束,从软件实现上来说,只是原来减一个值,现在减两个值,对系统性能影响很小。

(4) 动态调整哈希冲突

缓存 Cache 是指可以进行高速数据交换的存储器,由静态 RAM 组成,它先于内存与 CPU 交换数据,CPU 在运行时首先从一级缓存读取数据,若不命中再从二级缓存读取数据,仍不命中则从内存和虚拟内存读取数据,因此高速缓存的容量和速度直接影响到 CPU 的工作性能。目前所有主流 CPU 大都具有 L1 Cache(一级缓存)和 L2 Cache(二级缓存),少数高端 CPU 还集成了 L3 Cache(三级缓存)。L1 Cache 是 CPU 第一层高速缓存,分为指令 Cache 和数据 Cache,一般容量通常在 20 ~ 256kB 之间;L2 Cache 是 CPU 的第二层高速缓存,目前大都集成在片上,以同主频速度工作,一般容量在 128kB ~ 2048kB 之间。

根据对 WM 算法匹配过程的研究发现,在匹配过程中,每次 CPU 都要访问在预处理阶段建立的 Shift 表和 Hash 表,需要进一步匹配时再进行全特征匹配,因此,若能将这两张表全部放入片上 Cache 中,则可以大大减少访存次数,提高匹配性能,特别是在匹配较少的情况下,而 Shift 表和 Hash 表数据输入都是 4 字节 32 比特,数据输出位数决定了表的大小,同时也决定了冲突概率,位数越多,表越大,冲突越小;同理,位数越少,表越小,冲突越大。因此在算法运行时,根据不同 CPU 的 Cache 大小和不同的特征集规模,选定不同的 Shift 表和 Hash 表数据输出位数有利于提高系统整体性能,量化说明详见第 3 节的实验结果。

(5) 整数比较技术

在实际中,不同特征有相同后缀和前缀的概率并不大,基于此 WM 引进了 Prefix 表,当判断某一特征是否匹配时,先查 Prefix 表,若前缀匹配再进一步匹配特征非前缀部分。本文未使用 Prefix 表,这是因为进一步匹配是从特征头字节开始与文本匹配,实际匹配过程与 Prefix 表本质上一样。同时由于不区分前缀字符,整个特征可以统一处理,本文基于现代处理器处理带宽大都为 32 位的特点,以整数匹配操作替代原有的特征匹配,这样可以成倍减少匹配次数,同时在编程时利用 inline 技术减少函数调用的开销。

(6) 匹配跳跃属性

传统 WM 算法在跳跃值为 0 时进行精确匹配,判断完成后输入文本移动一个字节进行下一步判断,而实际上,即使在发生匹配的情形下,输入文本还是有可能跳跃的,比如说某个后缀只存在于某条特征中,那么,当确定此条特征是否匹配后,此 B 个字符并不存在于其他特征中,因此在偏移 $m - B + 1$ 字节以内是不可能匹配的,所以在 Hash 表中添加一项跳跃值,计算时,将全部特征的后缀 B 个字符不计算,其余类似于 Shift 表跳跃值计算方法。

3 ALPM 算法的性能评价

3.1 测试环境

为了评价改进算法的性能,基于开源项目 Snort[1]实现了 WM 算法、AC 算法,并实现了本文提出的 ALPM 算法,这 3 个算法具有统一的输入输出格式。本算法在进行性能测试时,采用随机生成特征集和测试数据的办法,计时采用 gettimeofday()函数,精度为 us 级,为了保证结果更为准确有效,采用比较大的测试文本(100MB)和 K 次最多测试方法,即程序连续运行 K 次,以结果中结果最接近的几次数据平均值为最终结果,这样可以尽可能保证数据的硬件无关性和结果可重复性,尽可能地体现算法性能。实验平台是一台服务器,CPU 是 Intel Core2 Duo E4400 2G,L2-Cache 大小是 2MB,主存是 DDR2-667 2GB,操作系统是 Windows XP SP2,算法开发平台为 Microsoft Visual Studio 2005。

3.2 性能评价

(1) 优化预处理流程

表 1 展示了 ALPM 算法采用优化技术处理预处理过程后的效果,可以看出,采用此技术增加了 Shift 表跳跃的概率,减少了访问 Hash 表后需要进一步匹配的平均特征数目,在增加不到一倍的预处理时间下,性能获得了 11% ~ 17% 的提升,而且特征集规模越大,性能提高越明显,同时此项优化技术不仅仅适用于本文所提算法,也可以用于经典 WM 算法的预处理过程中。

表 1　预处理过程优化效果表

特征规模	Shift 表为 0 的项		Hash 表平均链接特征数		预处理时间(ms)			性能(MB/s)		
	优化前	优化后	优化前	优化后	优化前	优化后	增加	优化前	优化后	提高
10k	9940	8878	1.04	1.03	18.8	20.1	6.90%	112.25	125.43	11.74%
30k	29458	23391	1.12	1.08	26.4	37.4	41.67%	89.24	100.5	12.62%
50k	48461	36262	1.2	1.12	36	62.8	74.44%	72.59	83.73	15.34%
70k	67005	48198	1.29	1.16	49.9	84.42	69.18%	55.43	64.38	16.15%
100k	93842	65494	1.43	1.22	68.3	105.94	55.11%	44.3	51.99	17.36%

(2) 动态调整哈希冲突

图 3 表示的是基于 Cache 大小和特征集规模动态调整哈希冲突的结果图,图中横坐标每项的两个数值分别表示 Shift 表和 Hash 表的地址位数,比如 20-18 表示 Shift 表包含 2^{20}项,Hash 表包含 2^{18}项,三条曲线分别表示特征集在 2 万、10 万和 20 万的匹配性能变化曲线。

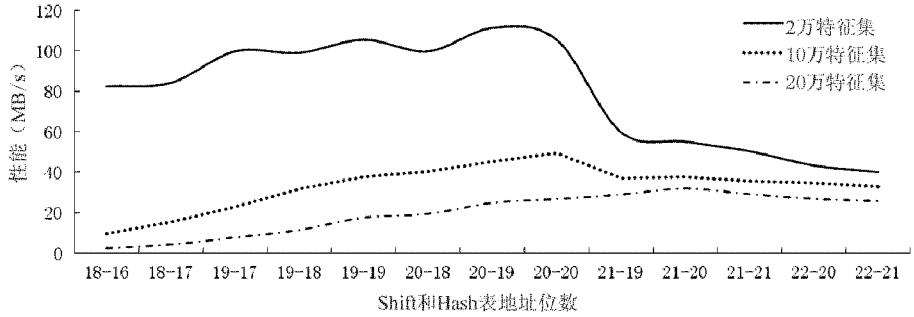

图 3　Shift 表和 Hash 表大小对匹配性能的影响

图 3 显示三条曲线在不同的配置下达到波峰,并且,特征集越大,波峰出现的越靠后。这是由于 Cache 大小和特征集规模都会影响 Shift 表和 Hash 表的访问效率,一方面,Shift 表和 Hash 表越小越容易放到 Cache 中,这样可以减少访存次数,提高算法性能,另一方面,Shift 表和 Hash 表越大哈希冲突越少,这样可以减少冗余匹配次数,同样可以提高算法性能,两者博弈的结果就是在中间某个配置点上算法获得最优性能;特征集越大哈希冲突越严重,越需要构建大的 Shift 表和 Hash 表,因此特征集越大,波峰出现的越靠后。实验测试证明,在实际中采用图 4所示的两个公式来动态调整 Shift 表和 Hash 表的大小是行之有效的。

$$Bits\_Shift \approx \log_2(\text{CacheSize}/2) + \lfloor numofPatterSet/100000 \rfloor$$

$$Bits\_Hash \approx \log_2(\text{CacheSize}/2) + \lfloor numofPatterSet/100000 \rfloor - 1$$

图 4　由 Cache 大小和特征集规模求 Shift 表和 Hash 表地址位数

(3) ALPM 算法匹配性能

ALPM 算法与经典的 WM 和 AC 算法的匹配性能比较见图 5 所示。随着特征集规模的不断增大,WM、AC、SBOM 等算法的性能急剧下降,在 2 万以上时都不超过 10MB/s,而 ALPM 算法性能呈亚线性下降趋势,并且在规模达到 20 万以上时仍能保持 30MB/s 的匹配速度。针对大规模特征集,ALPM 算法比经典算法快 5 ~ 10 倍。

(4) 多线程 ALPM 算法实现

随着计算机技术的发展,单芯片多处理器(single-chip mutiprocessor,CMP)、对称多处理(symmetrical multi-processing,SMP)等架构的服务器越来越普及,为了充分利用多核多处理器硬件资源,本文在单线程基础上设计并实现了多线程匹配算法。研究 ALPM 匹配过程不难发现,扫描过程中不会更改预处理时建立的 Shift 表和 Hash 表内容,因此,多线程实现方案将输入数据进行划分,每个线程处理一块数据,而共享预处理阶段建立的 Shift 表与 Hash 表,

需要注意的是切分数据时需要考虑切分时不能发生漏匹配。图 5 也显示了多线程版本的算法性能,通过充分利用处理器的双核资源,多线程算法性能基本上是单线程的 1.7 ~ 1.9 倍,随着硬件资源的增加,多线程版本可以获得更高的性能加速比。

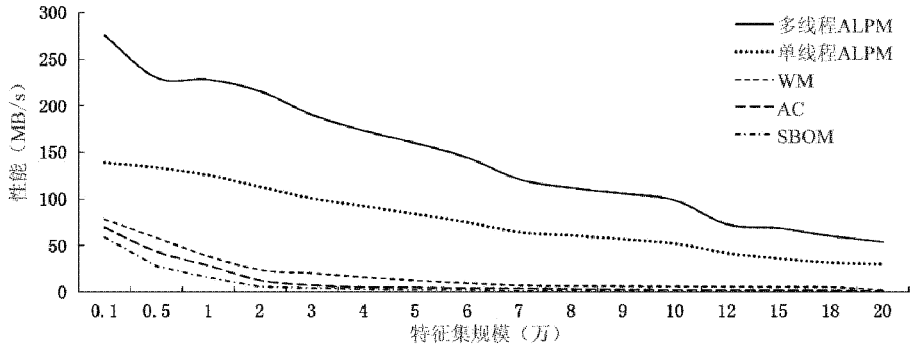

图 5 特征集规模对不同算法的影响

4 结 论

本文基于传统 WM 算法的跳跃思想,结合硬件结构特点,提出一种面向大规模特征集的高效特征匹配算法 ALPM,它对预处理过程和匹配过程分别提出了不同的优化策略。为支持大规模特征集,ALPM 首先将原 WM 算法每周期读入字符数由 2 字节提高到 4 字节,Shift 表和 Hash 表则采用不同的哈希函数来索引,并在预处理过程中动态截取特征标志,在匹配过程中结合 Cache 大小和特征集规模来共同调整哈希冲突概率,最后 ALPM 算法通过采用整数比较方法,增加匹配跳跃值属性等技术进一步提高算法性能。测试表明,针对大规模特征集,ALPM 算法匹配性能可比经典算法提高 5 ~ 10 倍。

参考文献

[1] Roesh M. Snort——lightweight intrusion detection for Networks. In: Proceedings of the 13th Systems Administration Conference, USENIX, Seattle, Washington, USA, 1999. 229-238

[2] ClamAV. Clam AntiVirus. http://www.clamav.net: ClamAV, 2002

[3] Fisk M, Varghese G. An analysis of fast string matching applied to content-based forwarding and intrusion detection: [technical report CS2001-0670]. San Diego: University of California-San Diego, 2002

[4] Jari K, Leena S, Jorma T. Tuning string matching for huge pattern sets. In: Proceedings of the 14th Annual Combinatorial Pattern Matching (CPM) Symposium, Morelia, Mexico, 2003. 211-224

[5] Aho A V, Corasick M J. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 1975,18(6):333-340

[6] Coit C J, Staniford S, McAlerney J. Towards faster string matching for intrusion detection or exceeding the speed of Snort. In: Proceedings of the DARPA Information Survivability Conference and Exposition Ⅱ (DISCEX'01), Los Alamitos,CA,USA, 2001.367-373

[7] Boyer R, Moore J. A fast string searching algorithm. *Communications of the ACM*, 1977,20(10):762-772

[8] Wu S, Manber U. A fast algorithm for multi-pattern searching: [technical report TR-94-17]. Tucson: University of Arizona,1994

[9] Blumer A, Blumer J, Ehrenfeucht A, et al. Complete inverted files for efficient text retrieval and analysis, *Journal of the ACM*, 1987,34(3):578-595

[10] Allauzen C, Raffinot M. Factor oracle of a set of words: [technical report 99-11]. Institute Gaspard-Monge,University de Marne-la-vallee, 1999

[11] Liu R T, Huang N F, Kao C N, et al. A fast string-matching algorithm for network processor-based intrusion detection system. *ACM Transactions on Embedded Computing Systems*, 2004, 3(3): 614-633

[12] 张鑫,谭建龙,程学旗. 一种改进的 Wu-Manber 多关键词匹配算法. 计算机应用, 2003, 23(7):29-31

[13] 宋华,戴一奇. 一种用于内容过滤和检测的快速多关键词识别算法. 计算机研究与发展, 2004, 41(6):940-945

[14] 刘萍, 谭建龙. XML 内容筛选中的快速串匹配算法. 中文信息学报, 2005, 19(2):20-27

[15] Aude L, Touzet H, Varre J S. Large scale matching for posi-

tion weight matrices. In: Proceedings of the 17th Annual Combinatorial Pattern Matching (CPM) Symposium, Barcelona, Spain, 2006.401-412

[16] Mathilde B, Dominique R, Stephane V. Longest Common Separable Pattern Among Permutations. In: Proceedings of the 18th Annual Combinatorial Pattern Matching (CPM) Symposium, London, Canada, 2007. 316-327

A fast string matching algorithm for large-scale pattern sets

Zhang Wei*, Xue Yibo**, Zhou Zongwei*, Wang Dongsheng***

(* Department of Computer Science and Technology, Tsinghua University, Beijing 100084)

(** Research Institute of Information Technology, Tsinghua University, Beijing 100084)

(*** Microprocessor and SoC Technology R&D Center, Tsinghua University, Beijing 100084)

Abstract

In view of the problem that the performance of the classical pattern matching (one of the key technologies for network and information security systems) algorithms degrades seriously when the patterns become large, especially over 50000, this paper proposes a new architectural large-scale pattern matching algorithm (ALPM) for large-scale pattern sets. Based on the shift concept of the classical Wu-Manber (WM) algorithm and combined with its features of hardware architecture, the ALPM adopts several pre-processing and matching strategies, such as utilizing two different Hash functions to access the Shift and hash tables, optimizing pre-processing to choose the best entry signs from patterns for the two tables and adjusting the Hash confliction dynamically with the Cache size and the pattern quantity, to improve the matching performance. The experimental results show that for the large-scale pattern set, the matching performance of the ALPM is 5 ~ 10 times higher than that of the classical WM.

Key words: large-scale pattern set, pattern matching, string matching, hash confliction, multi-threading