南京理工大学

硕士学位论文

开源网络入侵检测系统snort的检测算法研究

姓名：陈小茵

申请学位级别：硕士

专业：计算机应用技术

指导教师：兰少华

20071101

摘　要

检测引擎作为入侵检测系统(IDS)的核心模块，其检测速度快慢直接影响网络入侵检测系统的效率，模式匹配是入侵检测系统的重要检测方法，其性能对入侵检测系统至关重要。本文研究了入侵检测理论、开放源码的入侵检测系统 Snort 的架构及执行流程，详细分析了入侵检测系统常用得几种模式匹配算法，单模式 BM 算法、BMH 算法、QS 算法和多模式 AC 算法几种模式匹配算法，提出了一种改进的 HAC 多模式匹配算法，并分析了改进算法的时间复杂度和空间复杂度。改进算法可以同时进行多个模式的匹配，实现了在对文本一次扫描匹配过程中模式比较大的跳跃移动，可以更快地进行模式匹配。最后，对改进的算法进行了文本测试，实验表明改进的 HAC 算法在模式匹配速度方面的优势，可以用于网络入侵检测系统。

关键词：网络入侵检测系统　Snort　模式匹配　BM 算法　AC 算法

ABSTRACT

Detection moduleis the key module of the Intrusion Detection System(IDS). Pattern matching is a very important detection method for Network Intrusion Detection System, and it has a directly influence on the real time performance of the IDS. This paper firstly introduces the principle of IDS and frame of the Snort IDS, further analysis of the program. Then we study the main pattern matching algorithms now using in IDS such as BM, BMH, QS and AC, after that we bring forward an improved algorithm named HAC, and analyze both of their time complexity and pace complexity. The improved algorithm can match many patterns at one time and it obtains bigger skip value, so it can match patterns quicker. Finally some of these algorithms are implemented in text test, and experiments indicate that the improved algorithm HAC provides a significant improvement in pattern matching performance when it is used in an Intrusion Detection System.

KEY WORDS: Network Intrusion Detection System Snort pattern matching Boyer-Moore algorithm Aho-Corasick algorithm

声　明

本学位论文是我在导师的指导下取得的研究成果，尽我所知，在本学位论文中，除了加以标注和致谢的部分外，不包含其他人已经发表或公布过的研究成果，也不包含我为获得任何教育机构的学位或学历而使用过的材料。与我一同工作的同事对本学位论文做出的贡献均已在论文中作了明确的说明。

研究生签名：陈小茵　　2007年12月29日

学位论文使用授权声明

南京理工大学有权保存本学位论文的电子和纸质文档，可以借阅或上网公布本学位论文的部分或全部内容，可以向有关部门或机构送交并授权其保存、借阅或上网公布本学位论文的部分或全部内容。对于保密论文，按保密的有关规定和程序处理。

研究生签名：陈小茵　　2007年12月29日

1 绪论

1.1 课题提出背景

随着科学技术的发展，互联网给人类社会带来了前所未有变化，对经济、文化、生产、生活等人类社会各领域都有重大的影响，并逐渐成为人们日常工作和生活的一部分，如电子商务、远程教育、信息共享、休闲娱乐等都逐步走入我们的生活，影响着我们的生活。任何事物都具有两面性，互联网在给人类社会带来进步的同时，也存在许多问题。在这些问题中，计算机网络安全问题首当其冲，且日益显现，目前比较突出的安全问题是病毒事件侵袭和恶意软件的干扰。

据最新统计数据显示，各种网络安全事件每年都呈明显增长趋势，2007 年上半年，全国计算机感染台数 75，967，19 台，与去年同期相比增长了 12.2%[1]。随着网络用户和网络资源的大量增加，以及各种系统漏洞的大量存在和不断发现，使得网络安全问题变得更加错综复杂，并呈现出新的趋势[2]：（1）发现安全漏洞越来越快，覆盖面越来越广；（2）攻击工具越来越复杂；（3）攻击自动化程度和攻击速度提高，杀伤力逐步提高；（4）越来越不对称的威胁；（5）越来越高的防火墙渗透率；（6）对基础设施将形成越来越大的威胁。加之网络攻击行为日趋复杂，各种方法相互融合，使得网络安全防御更加困难。

对于这些安全问题，人们想出了各种各样的解决办法。在 2006 年，防火墙、防病毒以及入侵检测系统等“老三样”安全产品仍然占据了安全市场的大部分份额，也是目前比较流行的安全问题解决方案组合。防火墙是在内部网络与外部网络之间设置的一道屏障，它有效地阻止了未经许可的恶意的外部网络的访问，但是防火墙是一种静态的防御技术，它不会根据网络的变化来修改相应的防火墙规则，无法阻止协议漏洞发起的入侵，如蠕虫、垃圾邮件、病毒传播以及拒绝服务的侵扰。另外，防火墙本身的缺陷也是影响内部网络安全的重要问题，当黑客绕过防火墙攻击到内部时，防火墙形同虚设。入侵检测系统的作用是监控网络和计算机系统是否出现被入侵或误用的征兆。它被安置在防火墙之后。入侵检测系统不需要人工干预就可以不间断地运行，能够发现异于正常行为的操作。计算机安全防御领域的知名专家 Lance Spitzner 对安全防御问题做了比较形象的比喻：如果把计算机安全防御问题比喻为城堡，那么防火墙就可以作为城堡的护城河（只允许自己方面的队伍通过），入侵检测系统可以比作是城堡里的暗哨（监视是否有敌方、或是其他误入城堡的人出现）[3]，由此可见入侵检测系统系统的重要性。

1.2 国内外研究状况和发展趋势

1.2.1 几个相关概念

“入侵”是个广义的概念，美国韦氏大辞典是这样定义入侵的：“硬闯而入的行为，或是在没有受到邀请和欢迎的情况下进入一个地方。”[4]。入侵行为不仅来自外部，同时也指内部用户的未授权活动。入侵检测是指发现网络上一台计算机有未经授权的闯入行为。这个未经许可的访问或入侵，是一种对其他网络设备的安全威胁或伤害。入侵检测技术是为保证计算机系统的安全而设计与配置的一种能够及时发现并报告系统中未授权或异常现象的技术，是一种用于检测计算机网络中违反安全策略行为的技术[5]。入侵检测系统（Intrusion Detection System，简称 IDS） 是一套软件与硬件的结合体，是用来识别针对（网络）系统，或者更广泛意义上的信息系统的非法攻击，包括检测外界非法入侵者的恶意攻击或试探，以及内部合法用户的超越使用权限的非法行动。

1.2.2 IDS 发展历史

入侵检测的概念产生于二十年前。James P. Anderson 于 1980 年 4 月在一份题为 Computer Security Threat Monitoring and Surveillance（计算机安全威胁监控与监视）的技术报告中首次提出了 ID 的概念，他将入侵分为三类：外部攻击、内部攻击和误用行为，并提出了用审计记录来监视入侵活动的设想，但这一设想的重要性当时并未被理解。此文被认为是有关 ID 的最早论述[6]。1984 年到 1986 年乔治敦大学的 Dorothy Denning 和 SRI 公司计算机科学实验室的 Peter Neumann 提出了一种实时入侵检测系统模型——IDES（Intrusion Detection Expert Systems 入侵检测专家系统）。该模型有六个部分组成：主体、对象、审计记录、轮廓特征、异常记录、活动规则[7]。它独立于特定的系统平台、应用环境、系统弱点以及入侵检测类型，为构建入侵检测系统提供了一个通用的框架。该系统首次运用了统计和基于规则两种技术，是入侵检测研究中最有影响的一个系统。1987 年 Denning 女士在 IEEE 杂志上发表的 An Intrusion-Detection Model（一种入侵检测模型）文章，提出了一种通用的入侵检测模型。这篇文章也被认为是关于入侵检测技术的一篇最经典的文章。1989 年，加州大学戴维斯分校的 Todd Heberlein 写了一篇论文《A Network Security Monitor》，该监控器用于捕获 TCP/IP 分组，第一次直接将网络中的数据流作为入侵检测的审计数据来源，网络入侵检测从此诞生。至此之后，入侵检测系统发展史翻开新的一页，形成了两大研究阵营：基于网络的 IDS 和基于主机的 IDS。1988 年的莫里斯（Morris）蠕虫事件发生后，网络安全才真正引起了军方、学术界和企业的高度重视。从 20 世纪 90 年代到现在，IDS 的开发研制进入新时期，在智能化和分布式两个方向取得了长足的进展。1991 年，NADIR(Network Anomaly Detection and Intrusion Reporter)与 DIDS(Distribute Intrusion

Detection System)提出了收集和合并处理来自多个主机的审计信息从而用以检测针对一系列主机的协同攻击[8]。1994 年，Mark Crosbie 和 Gene Spafford 建议使用自治代理(Autonomous Agents)以便提高 IDS 的可伸缩性、可维护性、效率和容错性。1995 年，IDS 后续版本——NIDS(Next-Generation Intrusion Detection System)实现了可以检测多个主机上的入侵。 1996 年，GRIDS(Graph-based Intrusion Detection System)的设计和实现使得对大规模自动或协同攻击的检测更为便利。Forrest 等人将免疫原理运用到分布式入侵检测领域。1998 年，Ross Anderson 和 Abida Khattak 将信息检索技术引进到了入侵检测系统[9]。

在国内，入侵检测系统的研究和应用远远谈不到普及。究其原因，一方面是由于用户的认知程度低；另一方面是入侵检测在技术上还存在着困难。

1.2.3 入侵检测技术分类

根据着眼点的不同，入侵检测技术的分类方法有多种。根据数据来源的不同，IDS 可分为基于主机的入侵检测系统（HIDS）、基于网络的入侵检测系统（NIDS）和分布式入侵检测系统（DIDS）；根据数据分析方法的不同，可以分为误用入侵检测和异常入侵检测[10]。

基于主机入侵检测系统分析对象为主机审计日志，所以需要在主机上安装软件，安装配置较为复杂，占用主机资源，且对系统的运行和稳定性造成影响，目前在国内应用较少。基于网络入侵监测分析对象为网络数据流，只需安装在网络的监听端口上，对网络的运行无任何影响，目前国内使用较为广泛。

基于误用的入侵检测系统通过对比与已知的使用某种模式或者信号标识表示攻击是否相同进而发现攻击。这种方式可以检测许多甚至全部已知的攻击行为，对于未知的攻击手段却无能为力，而且对攻击特征描述正确性要求高，但是由于其易实现且较准确，目前已广泛应用于商用产品中。典型的误用检测系统是基于模式匹配的检测系统，本文研究的 Snort 就是这样的一个系统。

基于异常的入侵检测方法主要是基于这样的思想：任何正常行为都是有一定规律，可以通过分析这些行为产生的信息总结出一些规律，而入侵和滥用行为则通常和正常的行为存在严重的差异，通过检查这些差异就可以检测出入侵行为，此外不属于入侵的异常用户行为（如滥用自己的权限）也能被误认为是入侵[11]。学术上研究最多的是异常检测系统，它可以检测出未知的入侵，是未来研究的一个发展方向。目前研究较广泛的是基于神经网络的异常入侵检测和基于数据挖掘技术的异常检测系统。

对比误用检测与异常检测后可知，误用入侵检测是根据已知的系统或应用程序漏洞建立异常行为模型，然后将用户行为与之进行匹配，相同则为入侵。

因此，这种方法的优点是由于建模对象为已知的，所以可以得到较高的准确度；但是对于已知攻击的某些变体或是新型的攻击，它就无能为力了。而异常入侵检测正好相反，它是试图建立正常行为模型，任何违反该模型的事件的发生都被认为是可疑的，所以这种模型的好处是可以检测到一些未知的攻击，但是这种模型往往不能完全反映计算机系统的复杂的动态本质，因而很难建模。

目前，在世界范围内，误用入侵检测的应用比较流行。根据2005年全国计算机信息系统安全产品调查报告中指出，全国约有95%的入侵检测产品属于基于误用入侵检测模式来进行开发的，而在误用入侵检测中模式库的建立和模式匹配算法是其核心的所在[12]。

1.2.4 IDS中模式匹配算法研究现状

随着网络传输设备技术的长足进步，网络数据的传输速度呈几何级数增长，进而IDS的处理速度如果不能同步提高，将成为影响网络性能的一大瓶颈，虽然IDS通常以并联方式接入网络的，但如果其检测速度跟不上网络数据的传输速度，那么检测系统就会丢掉部分数据包以保证IDS的正常运行，从而导致漏报而影响系统的准确性和有效性。因此如何提高IDS的检测速度至观重要，而检测速度主要体现在对攻击信息的搜索速度，也即模式匹配的速度。选择合适的模式匹配算法，提高搜索速度是关键。

模式匹配的算法分类有多种，比较经典的单模式匹配算法有KMP（Knuth-Morris-Pratt）算法、Shift-And 或 Shift-Or算法、Boyer-Moore算法[13]、Horspool算法[14]、Quick Search算法[15]、BDM（Backward DAWG Matching）算法和BOM(Backward Oracle Matching)算法等。多模式匹配算法有Aho Corasick算法[16]、ommentz-Walter算法、Wu-Manber算法、SBDM（Set Backward DAWG Matching）算法以及SBOM（Set Backward Oracle Matching）算法等。

BM、BMH、QS算法匹配速度快，匹配过程中发生不匹配时，可以跳过部分不必要的比较。这种跳跃的策略对各种改进的算法有很大的启发。即利用多模匹配算法对文本的一次扫描可以找出所有模式的优势，结合跳过部分不必要比较的思想来提高算法的效率。有很多的研究者对多模AC算法结合单模BM、BMH、QS算法，在使AC算法每次匹配跳跃的距离更大方面都作了许多的研究。C. Jason Coit, Stuart Staniford和Joseph McAlerney在2001年在“Towards Fast String Matching for Intrusion Detection or Exceeding the Speed of Snort”[17]一文中提出了一种多模式匹配算法，即AC_BM算法，它实质上是将精确匹配的BM算法应用于多模式，允许由不同规则组成的一棵规则树同时对文本进行搜索匹配，也即AC算法和BM算法的组合使用，利用二者优点，效率同时优于AC算法和BM算法。2005年12月殷丽华，方滨兴在华中科技大学学报(自

然科学版)发表的一篇题为“一种改进的多模式匹配算法”[18]和 2007 年 6 月计算机应用杂志刊登的蔡晓妍，戴冠中，杨黎斌的一篇“改进的多模式字符串匹配算法”[19]，这两篇文章将单模式匹配算法中的 BMH 算法与经典的多模式匹配算法 AC 算法相结合。而 2002 年 1 月王永成、沈州、许一震在计算机研究与发展上发表的“改进的多模式匹配算法”[20]，2002 年 4 月许一震、王永成、沈州在上海交通大学学报上发表的“一种快速的多模式字符串匹配算法”[21]，是将单模式匹配算法中的 QS 算法与经典的多模式匹配算法 AC 算法相结合。几种算法均提出一个反向有限自动机的概念，在匹配过程中能够尽可能多地跳过待查文本串字符，并且在模式串长度较长或较短的情况下，均有相当好的性能。

从上面的论述中我们可以得到这样的结论，改进算法的不同之处大都在于一次跳过部分不必要比较的字符数量，即在失配后模式沿文本能移动的距离。在与自动机相结合的算法中，主要考虑失配后模式能够跳过多少个不需比较的文本字符。BM 算法是依据不匹配的文本字符来计算跳过的文本字符数；QS 算法是根据紧靠模式最右端的下一个文本字符来计算跳过的文本字符数；而 BMH 算法则是根据对齐模式最右端的文本字符来计算跳过的文本字符数。各算法都考虑到了跳跃，但都有其局限性，不可能每次都达到最大的跳跃距离。因此，本篇论文提出了一种新的模式匹配算法 HAC 算法，该算法混合 BM 算法、QS 算法、BMH 算法中的跳跃距离计算方法，结合 AC 算法，使 Aho-Corasick 自动机在对文本进行搜索的过程中每次均能保持 BM 算法、QS 算法、BMH 算法三种算法跳跃的最大距离，缩短比较次数和时间。该算法预处理阶段的时间复杂度为 $O\left(\sigma+\sum_{k=1}^{q} m_k\right)$，空间复杂度为 $O(\sigma)$，匹配阶段的最坏时间复杂度为 $O(n\cdot \max len)$。

1.2.5 IDS 研究的发展趋势

作为安全技术的一个重要领域，IDS 正在成为网络安全研究的热点领域，对 IDS 的理论研究和实际应用中还存在着许多问题，有待我们继续研究和探索。具体来说，入侵检测在以下方面有待继续发展[22][23]：

1. 在提高 IDS 的检测速度方面

随着网络传输设备技术的长足进步，网络数据的传输速度呈几何级数增长，而 IDS 的处理速度如果不能同步提高，将成为影响网络性能的一大瓶颈，虽然 IDS 通常以并联方式接入网络的，但如果其检测速度跟不上网络数据的传输速度，那么检测系统就会丢掉部分数据包以保证 IDS 的正常运行，从而导致漏报而影响系统的准确性和有效性。在 IDS 中，截获网络的每一个数据包，并分析、匹配其中是否具有某种攻击的特征需要花费大量的时间和系统资源，因此大部

分现有的 IDS 只有几十兆的检测速度，随着百兆、甚至千兆网络的大量应用，IDS 技术发展的速度已经远远落后于网络发展的速度。

2. 在提高 IDS 的安全性和准确性方面

目前 IDS 的商用领域还主要集中在基于模式匹配分析 IDS 和基于异常发现的 IDS。基于模式匹配分析方法的 IDS 在面对新的产生攻击方法和漏洞时，由于攻击特征库不能及时更新，或是很难总结出这些攻击的攻击特征等，都造成 IDS 漏报。而基于异常发现的 IDS 本身就导致了其漏报误报率较高。另外，大多 IDS 无法识别伪装或变形的网络攻击，也造成大量漏报和误报。在网络入侵检测系统的告警信息中仅有 10%左右是有用的，这也给网络管理带来负担。研究智能化入侵检测系统，即采用神经网络、遗传算法、模糊技术、免疫原理等方法，特别研究具有自学习能力的专家系统，实现知识库的不断升级与扩展，具有更广泛的应用前景。

3. 在提高 IDS 的协作性方面

实现网络安全是一项系统工程，不是一种网络产品单独就可以完成的，将网络安全作为一个整体工程来处理，设计全面的安全防御方案十分重要。从管理、网络结构、加密通道、防火墙、病毒防护、入侵检测多方位思考，共同协作来发现攻击、作出响应并阻止攻击。

4. 在 IDS 的理论研究方面

未形成一个比较完整的成熟的理论体系。随着计算机系统软、硬件的飞速发展，以及网络技术、分布式计算、系统工程、人工智能等计算机新兴技术与理论的不断发展与完善，入侵检测理论本身也处于发展变化中，但至今还未形成一个比较完整的成熟的理论体系。

5. 在 IDS 产品的规范方面

缺少国际、国内标准，产品规范不明确。标准化的工作对于一项技术的发展至关重要。在某一个技术领域，如果没有相应的标准，那么该领域的发展将是无序的。令人遗憾的是，尽管入侵检测系统经历了二十多年的发展，但目前尚无一个明确的国际标准出台，国内也没有，使得各产品之间无法共享数据信息。

毕竟入侵检测系统的发展只有短短的二十多年，还是一个新兴的科学领域，研究者们对入侵检测系统的未来还是充满信心。

1.3 论文的主要工作及论文的结构

本文主要研究网络入侵检测系统，从网络入侵检测系统的基本概念和发展动向着手，详细分析了开放源码的网络入侵检测系统 Snort 的工作模式、系统架构、规则检测，并给出了 Snort2.0 以上版本的快速匹配规则引擎的详细解释，

对其检测引擎采用的模式匹配算法进行了深入研究，分析解释了经典单模式字符串匹配算法和多模式字符串匹配算法，在分析研究基础上提出了改进的 HAC 算法思想及实现，最后对几种算法进行了实验比较。

论文全文章节安排如下：

第一章介绍了论文提出背景、入侵检测系统国内外研究概况及发展趋势、模式匹配算法研究现状，及论文的结构安排。

第二章介绍了开源网络入侵检测系统 SNORT 的总体结构和规则，并对 Snort 的流程进行了详细分析。

第三章对入侵检测系统中的模式匹配算法的选用进行了研究，详细分析了 BM 单模式匹配算法和 AC 多模式匹配算法，在此基础上提出了改进的多模式匹配 HAC 算法，并对算法的性能进行了分析。

第四章对 HAC 算法进行了性能测试，证明了改进后的多模式匹配算法可以提高入侵检测系统的效率。

第五章对全文作了总结，并对本文的研究工作提出了进一步的研究方向。

2 开源入侵检测系统 SNORT

2.1 SNORT 简介

2.1.1 基本情况

Snort是一个用C语言编写的开放源代码软件[24],该软件遵循通用公共许可证GPL（GNU General Public License）的要求，只要遵守GPL的任何组织和个人都可以自由使用。1998年12月22日Snort被放到了Packet Storm网站上供下载。Snort的适应性很强，在全世界范围内被广泛的安装和使用。

Snort是一个跨平台、轻量级网络入侵检测系统。其工作原理为基于共享网络上原始的网络传输数据，通过分析捕获的数据包，匹配入侵行为的特征或从网络活动的角度检测异常行为，进而采取入侵的预警或记录。从入侵检测分类上来看，Snort应该算是一个基于网络和误用的入侵检测软件。针对每一种入侵行为，都提炼出它的特征值并按照规范写成检验规则，从而形成一个规则库，将捕获的数据包对照规则库逐一匹配，若匹配成功，则认为该入侵行为成立。

许多商业运营的入侵检测系统都是在Snort检测引擎的基础上进行二次开发的。历经数年发展，Snort已经发展到2.7版本，其基本架构在1.6版本时初步建立，2.0版本中采用了新型的架构设计和入侵检测方法，功能更加完善，部署也变得更加方便，Snort系统已经成为学习研究入侵检测的经典实例。

2.1.2 特点

跨平台、轻量级网络入侵检测系统 Snort 具有以下特点[25]：

1. Snort 具有跨平台性,它支持的操作系统广泛。支持 Windows 系列、Linux 系列、SunOS、 Solaris、MacOS X server、OpenBSD、HP-UNIX 等。

2. Snort 是轻量级网络入侵检测系统，所谓轻量级是指系统管理员可以轻易地将 Snort 安装到网络中去，可以在很短的时间内完成配置，可以很方便地集成到网络安全的整体方案中，使其成为网络安全体系的有机组成部分。

3. 具有实时流量分析和记录 IP 网络数据包的能力。能够快速检测网络攻击，及时发出警告。它提供的警告方式很丰富，比如 syslog，UnixSocket，winPopup 等。

4. Snort 能够进行协议分析，内容的匹配和搜索。现在它能够分析的协议有 TCP, UDP 和 ICMP。将来的版本，将提供对 ARP，ICRP，GRE，OSPF，RIP，ERIP，IPX，APPLEX 等协议的支持。它也能检测多种方式的攻击和探测，比如：缓冲区溢出，CGI 攻击，SMB 检测，探测操作系统特征的企图等。

5. Snort 具有灵活的日志格式。支持 Tcpdump 的二进制格式,也支持 ASCII 字符形式，也支持 XML 格式的，更便于维护和检查。使用数据库输出插件，Snort 可以把日志记入数据库，当前支持的大多数流行数据库，包括：MySQL、Microsoft

SQL Serve 以及 Oracle、和 UnixODBC 等。

6. 使用 TCP 流插件（TCPSTREAM），Snort 可以对 TCP 包进行重组，这种能力使得 Snort 可以对抗“无状态”攻击。无状态攻击是指攻击者使用一个程序，每次只发送一个字节的数据包，就完全可以避开 Snort 的模式匹配，而被攻击主机的 TCP 栈会重新组合这些数据包，将攻击数据发送给目标端口上监听的进程，从而使攻击包逃过 Snort 的检测。使用 TCP 流插件，可以对 TCP 包进行缓冲，然后进行匹配，使 Snort 具备对付上面攻击的能力。

7. 使用 Spade（Statistical Packet Anomaly Detection Engine）插件，Snort 能够报告异常的数据包，从而对端口扫描进行有效的检测。

8. Snort 还具有很强的系统防护能力。使用 IPTables，IPFilter 插件可以使入侵检测主机和防火墙联动，通过 FlexResp 功能，Snort 能够命令防火墙断开恶意连接。

9. Snort 扩展性好，对于新的攻击威胁反应迅速。Snort 采用了一种简单的规则描述语言（很多商用入侵检测系统都兼容 Snort 的规则语言）。最基本的规则包含四个域：处理动作，协议，方向，端口，例如：Log Tcp Any any->10.1.1.0/24 80。

10. Snort 的规则语言非常简单，能够对新的网络攻击做出很快的反应。发现新攻击后，可以很快地根据 Bugtrag 邮件列表，找到特征码，写出新的规则文件，迅速建立规则表。

11. Snort 支持插件，可以使用具有特定功能的报告，检测子系统插件对其进行功能扩展。Snort 当前支持的插件有：数据库日志输出插件，破碎数据包检测插件，端口扫描检测插件，Http URL 插件，XML 网页生成插件等。

总之，对于世界上各个安全组织来讲，Snort都是一个优秀的入侵检测系统的标准。通过研究它，可以学到入侵检测系统的内部框架及工作流程(也包括同类型的商业入侵检测系统的框架及工作流程)。

2.1.3 协议分析技术简介

协议分析技术的特点是充分利用网络协议的高度有序性，即网络协议并非随机变化的字节流，而是高度有序的系统，协议包中的数据结构应该是完全可知的，且与一系列协议规则密切关联[26]。

Snort系统的入侵检测就是基于协议分析技术基础的模式匹配。充分利用协议的有序性来快速检测某个攻击特征的存在，从而大大降低了特征搜索所需的计算量。下面以一个具体的数据包的例子来说明协议分析的处理过程。

协议规则规定，以太网数据包在第13个字符处开始，包含了2B（字节）的网络层协议标示。协议解码软件利用这一信息指示第一步的检测工作，即忽略前12

个字符，直接跳到第13个字符位置，并读取2B的协议标示。见图2.1.3。

第一步，直接跳到第 13 个字符位置，并读取 2B 的协议标示（0800）。

如果检测到的是0800，则说明该以太网帧携带的是IP包，协议解码函数利用这一信息指示第二步检测工作，利用IP协议规则，即IP包在第24个字符处开始，包含1B的传输层协议标示。因此协议解码将忽略掉从第17到第23个字符，直接跳到第24个字符处读取1B的传输层协议标示。

AF7*Hy289s82	0800	B9v5yt$	06	11tbhk765	00801293ugdB2%00397e	39123456789……

图2.1.3.1 协议解码图

第二步，跳到第 24 个字符处读取 1B 的传输层协议标示（06）。

如果读到的值是06，则说明这个IP帧携带的是TCP包，协议解码函数利用这一信息指示第三步检测工作，利用TCP协议规则，即TCP包在第35个字符处开始，包含了一对2B的应用层协议标示，这对2B的应用层协议标示一般被当成“端口号”。因此，协议解码将忽略掉从第26到第34个字符，直接跳到第35个字符处读取4B的2个端口号。

第三步，跳到第 35 个字符处读取一对端口号。

如果有一个端口号是 0080，则说明这个 TCP 帧携带的是 HTTP 数据包，协议解码函数利用这一信息指示第四步检测工作。

第四步，解析器从第 55 个字符处开始读取 URL 字符串。

到此为止，协议解码软件已经找到了应用层数据的位置，并且已知应用协议是 HTTP，并且从 TCP 包第 55 个字符处开始，是 URL 信息。该 URL 字符串将被提交给 HTTP 解析器，进行进一步的处理工作。

2.2 SNORT 的规则

Snort 的规则文件就是 Snort 的攻击知识库，每条规则包含一种攻击的攻击标识，Snort 就是通过规则来识别攻击。这些规则被分类放在 Snort 源码所在目录*.rules 文件中，如 web.rules。从 Snort 网站下载的最新规则 snortrules-pr-2.4.tar 表明 Snort 的规则已增加到了 4000 多条。

2.2.1 Snort 的规则

Snort 规则使用“一种简单的、轻量级的描述语言”来描述网络上带有攻击标识的数据包。这种描述语言在形式上不够完备，但相当灵活，而且具有很强的描述能力。

Snort 中一条完整的规则在逻辑上分为两部分：

规则头（Ruler Head）和规则选项（Rule Option）[7]。规则头是特征中的重要部分，它定义了规则被触发时的行为、对应的网络报文的协议以及包括 IP 地址、网络、端口在内的源信息和目的信息。通过这些限制，规则或者说规则

选项需要处理的数据量可能会被有效地减少。

规则选项部分包含了所要显示给用户查看的报警信息及用来判断此报文是否为攻击报文的检测信息。

下面是一条简单的规则：

alert tcp any any->192. 168. 1. 0/24 111 (content:"|00 01 86 a5|";msg:"mountd access";)

注：从 1.8 版本开始，Snort 允许一条规则可以分行写，行尾用"\"作为分行的标识。

在这个例子中，圆括号左面是规则头，括号中间的部分为规则选项，规则选项中冒号前的部分称为选项关键字（Option Keyword），需要注意并不是每条规则都必需要有规则选项部分，它只是用来更好地定义出表示某种攻击、需要采取某种行为的报文。图 2.2.1.1 表示规则的内容。只有当规则中的每一个元素都为真时，才能触发对应的规则动作。综合起来考虑，规则元素之间形成的是一种逻辑"与"的关系；同时，规则库文件中的各条规则之间形成的是一种更大范围上的逻辑"或"关系。[26]

alert tcp any any->192. 168.1.0/24 111	(content:"|00 01 86 a5|";msg:"mountd access";)
规则头	规则选项

图2.2.1.1　规则内容

2.2.2 规则头

规则头定义了报警的类型和哪一种协议、IP地址和IP协议端口将会被监控。规则头由规则行为、协议字段、地址和端口信息3部分组成。

Alert	tcp	any	any	→	192. 168.1.0/24	111
动作	协议	源IP	源端口	攻击方向	目的IP	目的端口

图2.2.2.1 规则头内容

1．规则行为

规则头的第一个字段就是规则行为。Snort定义了五种可选的行为：alert，log，pass，activate，dynamic。其语义如下：

Alert：使用设定的警告方法生成警告信息，并记录这个报文。

Log：使用设定的记录方法记录这个报文。

Pass：忽略这个报文。

Activate：进行 alert，然后激活另一个 dynamic 规则。

Dynamic：等待被一个 activate 规则激活，然后进行 log。

2．协议字段

规则头的下一个域是协议字段。当前Snort支持三种IP上的协议—TCP、UDP和ICMP，将来可能会支持更多的协议。这个字段中可能的值包括TCP、UDP和ICMP。

3．地址和端口信息

规则头的下一部分是描述规则的 IP 地址和端口信息。

格式为：IP/CIDR Port

地址部分必须使用 IP 地址加上一个 CIDR（Classless Inter-Domain Routing）块，CIDR 块用来说明 IP 地址的网络掩码，/24 表示一个 C 类地址，/16 表示一个 B 类地址，/32 指定一个主机。如 IP 地址和 CIDR 块结合，192.168.1.0/24 就表示从 192.168.1.0 到 192.168.1.255 的 IP 地址范围。

负操作符“!”可以应用到 IP 地址上，表示除了此 IP 地址范围外的所有地址。

IP 地址字段也可以是 IP 地址列表，将 IP 地址列表置于方括号内，使用逗号分隔 IP/CIDR 块。例如：![192.168.1.0/24,10.1.1.0/24]

表示除了从 192.168.1.0 到 192.168.1.255 的 IP 地址范围和从 10.1.1.0 到 10.1.1.255 的 IP 地址范围外的所有 IP 地址范围。

端口号部分可以使用很多方法来说明：

关键字 any 说明任意端口。

单个数字指定静态的端口。

使用冒号隔开的两个数字表示端口的范围。例如：1：1024 表示从 1 到 1024 的端口。

同样可以使用负操作符“!”。

还可以使用方向操作符来限制数据报文的流向。

“->”表示从左端流向右端的数据报文

“<>” 表示双向的数据流。

2.2.3 规则选项

在规则的圆括号中的部分是规则选项，规则选项的作用是在规则头信息的基础上作进一步的分析，有了它才能确认复杂的攻击。规则选项定义了攻击数据包的特定特征，它形成了检测系统的核心，功能强大，具有易用性和灵活性，入侵检测系统根据规则选项检查网络数据包。一个规则的规则选项中可能有多个选项，不同选项之间使用“；”分隔，规则选项的关键字和它的值之间使用冒号分隔。

Snort 选项有三十几种关键字，常用的选项关键字如下：

msg：打印一条警告信息到警告或日志中。

content：在报文负载中搜索某个模式。这个字段是 Snort 的一个重要特征。进行比较的数据可以包含二进制和 ASCII 数据，二进制数据使用“|”括起来的十六进制字节串表示，如：Content:”|90C8 COFF FFFF|/bin/sh”。除非指定 nocase，否则这种匹配是大小写敏感的。另外需要注意的是这种比较非常耗费资源，一定要增加尽可能多得其他选项来限制进行搜索的报文。

content-list：在报文负载中搜索一个模式的集合。

offset：调整 content 选项，设置模式匹配开始的偏移量。

depth：调整 content 选项，设置模式匹配的最大搜索长度。

nocase：进行内容匹配时，字符串的大小写不敏感。

uricontent：允许对特定的 Web 协议进行通信分析。它只匹配 URI 请求部分。

另外还有许多检查协议域的关键字，如ttl：检查IP报文的TTL域的值；id：检查IP报文的分片ID域的值等等。

2.3 Snort 的总体结构分析

Snort 是一个非常优秀的、开放源代码的入侵检测系统：

从检测机制上看，它不仅具有基于规则的误用检测方法，而且还有基于异常的检测方法（预处理功能）。

从体系结构上看，它充分考虑到扩展的需求，大量使用了插件机制。

从功能模块上看，各个模块功能明晰，相对独立，设计合理。

从编码上看，具有很好的设计风格和详细的注释，易于理解。

2.3.1 总体结构

Snort 的体系结构功能模块设计独立，功能明晰。下图图 2.3.1.1Snort 总体模块图[5]为 Snort 模块的组成以及相互关系。

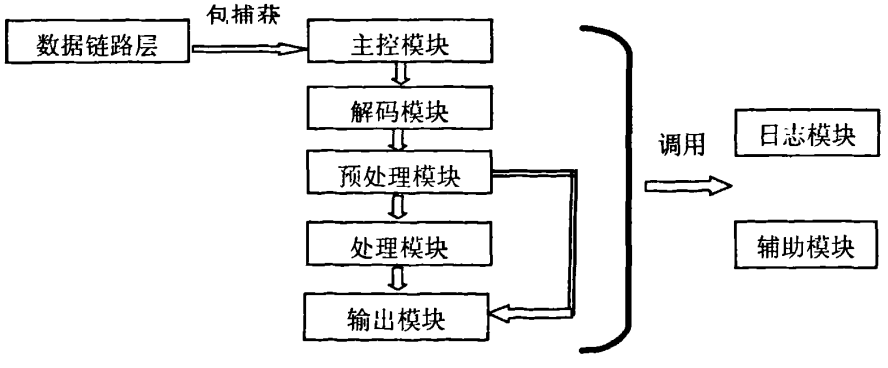

图2.3.1.1　Snort总体模块图

下面就从功能上对图 2.3.1.1 中的各个模块进行分析说明：

主控模块实现的功能包括所有模块的初始化、命令行参数解释、配置文件

解释、数据包捕获库 Libpcap 的初始化，然后调用 Libpcap 开始捕获数据包，初始化插件链表，初始化预处理插件，读入规则，形成规则匹配数据结构，规则快速匹配数据结构和输出数据结构。

解码模块把从网络上抓取的原始数据，从下向上沿各个协议进行解码并填充相应的数据结构，以便规则处理模块处理。

预处理模块在处理模块之前进行，在对报文进行模式匹配之前，先进行分片重组、流重组和异常检查等预处理操作。

处理模块实现对这些报文进行基于规则的模式匹配工作，检测出入侵行为；在初始化阶段它还负责完成规则文件的解释和规则语法树的构建工作。处理模块主要检查数据包的各个方面，包括数据包的大小、协议类型、IP/ICMP/TCP/UDP 的选项等，辅助规则匹配完成检测功能。

输出模块实现在检测到攻击后执行各种输出和反应的功能。

日志模块实现各种报文日志功能，也就是把各种类型的报文记录到各种类型的日志中。

在系统运行的过程中，还使用一些辅助模块；树结构定义子模块定义了几种 Snort 使用的二叉树结构和相关的处理函数，tag 处理子模块完成了和 tag 相关的功能，另外一些子模块也提供了一些公用的函数，如字符串处理等。

2.3.2 Snort 源文件分组说明

按图 2.3.1.1 所示的 Snort 总体结构中的各模块，可以把 Snort2.7 软件包中的文件作一个分组：

1. 主控模块

snort.c(h) 是主程序所在文件，实现了 main 函数和一系列的初始化函数。

plugbase. c(h)实现了初始化检测和注册检测规则的一组函数。Snort中的检测规则以链表的形式存储，每条规则通过注册过程添加到链表中。

2. 解码模块

decode.c(h)完成报文解码过程，把网络数据包解码成 Snort 定义的 packet 结构，用于后续分析。

3. 规则处理模块

Rules.h 定义了生成二维规则链表的各种类型变量的数据结构。

Parser.c(h)规则解析。

detect.c(h)处理规则头信息，构造规则链表

pcrm.c(h)构造快速匹配的规则链表的辅助函数。

fpcreate.c(h)构造快速匹配的规则链表

fpdetect.c/(h)用于快速规则匹配检测

4. 预处理插件模块

保存在\preprocessors 子目录中的文件。实现 http 解码、数据包分片检查和端口扫描检测等。

5. 处理插件模块

保存在\detection-plugins 子目录中的文件。实现不同类型的检测规则。可以很容易的从文件名得知所实现的规则，例如：sp_dsize_check.c 针对的是包的数据大小，sp_icmp_type_check.c 针对的是 icmp 包的类型，sp_tcp_flag_check.c 针对的是 TCP 包的标志位，还有规则选项的模式匹配文件 sp_pattern_match.c 等等。

6. 输出插件模块

保存在\output-plugins 子目录中的文件。实现输出规则，以不同的方式记录事件。例如 syslog、tcpdump 等。

7. 日志模块

Log.c(h)实现日志和报警功能。

8. 辅助模块

UbiBinTree.c(h)实现一个简单的二叉树。

ubi_SplayTree. c (h) 实现了一个伸展的二叉树和相关的功能。

tag. c (h)实现与 tag 有关的高级日志操作。

mstring.c(h)实现字符串匹配 Boyer-Moore 算法。

strlcatu.c (h)文件只有一个函数 strlcat(dst, src, siz)，实现把 src 字符串追加到 dst 的后面，siz 用来限定 dst 的最终长度。

strlcpyu.c (h) 文件只有一个函数strlcpy (dst, src, siz)，实现把src字符串拷贝到dst，siz用来限定复制的长度。

snprintf.c (h)定义了一些增强的输出函数，由 configure 决定是否使用。

codes.c(h)定义了unicode_entry结构以及unicode_entry类型的数组unicode_data，该数组包含了建立unicode与ASCII码之间的映射所需的数据。

debug.c(h)定义了debug的级别和GetDebugLevel函数获取相应得Debug级别，DebugMessageFunc函数确认Debug参数变量的格式化输出。

保存在\ sfutil子目录中的文件。实现多模匹配算法，如AC、WMN等。

2.4 Snort 流程概要分析

2.4.1 初始化模块

Snort系统首先执行命令行参数解析函数ParseCmdLine，根据系统配置，设置系统参数，包括规则文件、系统运行模式、显示模式、插件激活模式等。

调用检测引擎初始化函数fpInitDetectionEngine，为快速规则匹配模块配

置缺省参数，包括缺省模式匹配算法等。

插件初始化，包括预处理插件初始化、处理（检测引擎）插件初始化和输出插件初始化等。各种插件的初始化过程都是类似的，主要任务就是将各种插件的关键字名称和对应的初始化处理函数联系起来，并注册到对应的关键字链表结构中。采用插件机制，是Snort的一大特色，它可以轻松实现入侵检测的扩展。

在InitPlugIns()函数中调用各种处理模块、预处理模块、输出模块的配置函数，如SetupPatternMatch()模式匹配函数设置、SetupDsizeCheck()检查数据包大小函数设置等。在SetupPatternMatch()中，又调用RegisterPlugin(“content”, PayloadSearchInit)函数，对规则的关键字和调用的处理函数进行了关联，并注册到对应的关键字链表结构中。如规则选项关键字”content”对应处理函数PayloadSearchInit。

以检测引擎插件初始化为例，对应的关键字链表结构 KeywordXlateList 如下：

```
typedef struct _KeywordXlate
{
   char *keyword;
   void (*func)(char *, OptTreeNode *, int);
} KeywordXlate;
typedef struct _KeywordXlateList
{
   KeywordXlate entry;
   struct _KeywordXlateList *next;
} KeywordXlateList;
```

检测引擎插件初始化函数void (*func)(char *, OptTreeNode *, int)并非插件的直接工作模块，而是由AddOptFuncToList函数负责调用，在规则解析中解析插件参数，然后将对应的插件处理模块链入到规则检测函数链表结构中。

插件初始化结束，得到三个链表指针KeywordXlateList、 PreprocessList和OutputKeywordList。这三个链表指针都是plugbase.c文件中的全局变量，且都是对应的OutputKeyNode/PreprocessNode/KeywordXlate结构的链表头，这些结构都在plugbase.h头文件中定义，它们都包括用来查找的插件名和要用到的处理从规则配置文件中传过来参数的插件初始化函数。具体要用到那些插件是在parser.c文件中ParseRule函数实现。

数据包捕获函数

Snort利用libpcap(用于捕获网络数据包的必备工具，它是独立于系统的API

接口，为底层网络监控提供了一个可移植的框架，可用于网络统计收集、安全监控、网络调试等应用）从网卡捕获网络封包。下面是数据包捕获函数的执行情况。

OpenPcap()：根据命令行参数的解析结果，OpenPcap()函数依次调用图2.4.1.1基于Libpcap应用的基础流程[10]的libpcap库中的函数。循环抓包pcap_loop函数在snort.c主流程的InterfaceThread函数中实现。

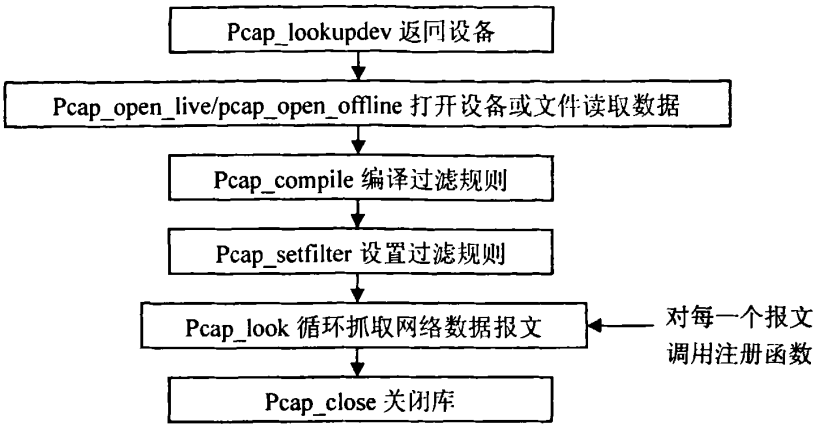

图2.4.1.1 基于Libpcap应用的基础流程

2.4.2 数据包解码模块

数据包解码模块将数据包根据协议层次进行解析，并填充到 Packet 包结构中，为进一步的分析提供数据。

1.数据包结构

Packet 包结构在 decode.h 文件中定义。它定义了抓包的 BPF(Berkeley Packet Filter)数据指针，以及各种协议报文所需的数据结构，如 Token Ring 令牌环网、Ethernet 以太网、无线局域网、802.1x 协议、PPPoE 协议等等所需的报文数据结构，如针对 Ethernet 以太网的 TCP 协议，就定义了：

EtherHdr *eh——标准以太网 TCP/IP/ARP 的报头指针；报头包括源 IP，目的 IP 和报文类型。

TCPHdr *tcph, *orig_tcph——TCP 报头指针；

u_int32_t tcp_options_len——TCP 报头内容的长度

u_int8_t *tcp_options_data——指向 TCP 报头内容的指针

u_int8_t frag_flag——分片数据包标志

u_int16_t frag_offset——分片偏移量

u_int16_t sp——源端口(TCP/UDP)

u_int16_t dp——目的端口(TCP/UDP)

u_int16_t orig_sp——初始分片源端口(TCP/UDP)

u_int16_t orig_dp——初始分片目的端口(TCP/UDP)

u_int32_t caplen——抓包长度

等等。

2. 数据包解码

snort.c 文件中的 main 函数在启动时，首先执行 SetPktProcessor()初始化工作，SetPktProcessor()函数根据检测到的接口类型 datalink(由 libpcap 库函数得到）的值判断网络类型。

```
switch(datalink)
{
   case DLT_EN10MB:              /* Ethernet 以太网*/
   grinder = DecodeEthPkt;       //指明解码函数
   break;
   case DLT_IEEE802:          /* Token Ring 令牌环网*/
   grinder = DecodeTRPkt;
   break;
}
```

例如：当发现一个接口为以太网接口时，指明使用 DecodeEthPkt 函数解码。

SetPktProcessor()初始化工作结束时，已经知道调用哪个解码函数了。

再调用decode.c文件中的各种解码函数对数据包进行协议解码，将解析后的结果填充到Packet数据结构中，供后续模块调用。解码函数有：DecodeEthPkt()（以太网解码）、DecodeIEEE80211Pkt()（无线网络解码）等，下面以以太网的解码函数DecodeEthPkt()调用DecodeIP()，再调用DecodeTCP()为例，解析解码过程，其它类型网络的解码程序类似。

DecodeEthPkt（）以太网解码函数，参数有三个参数：p=>指向数据包结构的指针，pkthdr=>指向libpcap抓包函数指针，Pkt=>指向真实的数据包头部指针。

```
DecodeEthPkt(Packet * p, struct pcap_pkthdr * pkthdr, u_int8_t *
pkt)
{
   switch(ntohs(p->eh->ether_type))
   {
      case ETHERNET_TYPE_IP:
         DecodeIP(p->pkt + ETHERNET_HEADER_LEN,
         cap_len - ETHERNET_HEADER_LEN, p);
      return;
   }
```

```
}
```

根据不同的数据报文类型，调用不同的函数继续解码。数据报文的类型在以太网的报头中表明(p->eh->ether_type)，这个值如果是0x0800，则调用DecodeIP函数。

注意，在调用DecodeIP时，p->pkt指针向后移动了ETHERNET_HEADER_LEN(14)位，指向数据报的IP报头。长度也变为cap_len - ETHERNET_HEADER_LEN，去掉了以太网报头的长度。

DecodeIP ( ) IP网络协议解码函数，有三个参数：pkt =>数据包指针，len => 数据包长度，p =>指向包结构的指针。

```
void DecodeIP(u_int8_t * pkt, const u_int32_t len, Packet * p)
{
   ip_len = ntohs(p->iph->ip_len);/* 设置 IP 数据报长度 */
   hlen = IP_HLEN(p->iph) << 2;/* 设置 IP 报头长度  */
   switch(p->iph->ip_proto)
   {
      case IPPROTO_TCP:        /*UDP、ICMP 等协议解析与此类似*/
      pc.tcp++;
      DecodeTCP(pkt + hlen, ip_len, p);
      return;
      ......
   }
}
```

把IP报头中的内容填充到包结构中，如offset、mf、df等。如果IP报头显示为TCP数据包（p->iph->ip_proto），调用DecodeTCP函数继续解码。此处的pkt指针一样要后移（pkt + hlen），跳过IP报头，指向下面的TCP报头。

DecodeTCP ( ) TCP传输协议解码函数，参数有三个：pkt =>数据报文指针，len =>数据报长度，p =>解码数据结构指针。在函数中判断是否有TCP选项，有，则调用DecodeTCPOptions函数对TCP选项解码，然后有4项重要的变量被赋值：sp为数据包源端口；dp为数据包目的端口；data为指向数据包数据段的指针；dsize为数据包的数据段长度。这几项内容被后面的预处理和规则处理模块广泛使用。

至此，数据包解码过程结束，解析后的结果填充到 Packet 数据结构中。

2.4.3 预处理模块

Snort 系统在初始化（主控模块中）完成后，进入解码模块，解码完成后需要对数据包进行预处理。请注意 SNORT 使用的插件的思想，Snort 的插件结

构允许开发者扩展 Snort 的功能。

预处理 Preprocess 函数遍历在初始化预处理插件后得到的 PreprocessKeywordList 链表，调用配置文件中要求的预处理插件函数对所抓的数据包进行预处理。形成预处理处理关键字链表，并将关键字和预处理函数关联。

预处理模块在调用检测引擎之前，在数据包被解码之后运行。通过这种机制，Snort可以以一种特殊的方式对数据包进行修改或者分析。对预处理器的调用和配置使用preprocessor关键字，其格式为preprocessor<name><options>。以下是几个常用的预处理插件[26]：

1. HTTP解码预处理模块用来处理HTTP URI字符串，把它们转换为清晰的ASCII字符串。这样就可以对抗evasice web URL扫描程序和能够避开字符串内容分析的恶意攻击者。

2. frag2模块使Snort能够消除IP碎片包，给黑客使用IP碎片包绕过系统的检测增加了难度。

3. stream4插件为Snort提供了TCP数据包重组的功能。在配置的端口上，stream4插件能够对TCP数据包的细小片段进行重组成为完整的TCP数据包，然后Snort可以对其可疑行为进行检查。

4. Portscan预处理程序的用处： 向标准记录设备中记录从一个源IP地址的端口扫描的开始和结束。端口扫描可以是对任一IP地址的多个端口，也可以是对多个IP地址的同一端口进行。可以处理分布式的端口扫描（多对一或多对多）。端口扫描也包括单一的秘密扫描(stealth scan)数据包，比如NULL,FIN,SYNFIN,XMAS等。

5. Portscan2模块将检测端口扫描。它要求包含Conversation预处理器以便判定一个会话是什么时间开始的。它的目的是能够检测快速扫描，例如，快速的nmap扫描。

6. Conversation预处理器使Snort能够得到关于协议的基本的会话状态而不仅仅是由spp_stream4处理的TCP状态。当它接收到一个你的网络不允许的协议的数据包时，它也能产生一个报警信息。要做到这一点，请在IP协议列表中设置你允许的IP协议，并且当它收到一个不允许的数据包时，它将报警并记录这个数据包。

7. Http Flow 模块可以忽略 HTTP 头后面的 HTTP 服务响应。

8. 异常检测：在 Snort 误用检测框架实现异常检测的功能。

2.4.4 处理模块

规则处理模块主要分为以下几步：规则解析，构造规则链过程，构造快速匹

配规则链表。

1. 规则链表的数据结构

Snort的主要数据结构就是几个链表。Snort按照规则的处理动作（alert、pass、log等）来划分成几个链表。其中每个链表又按照协议类型TCP、UDP和ICMP分成三个链表，所有的规则都会被分配到这几个链表中。描述每条规则的数据结构是RuleTreeNode（RTN）和OptTreeNode（OTN）。图2.4.4.1是规则的内存表示层次结构图。

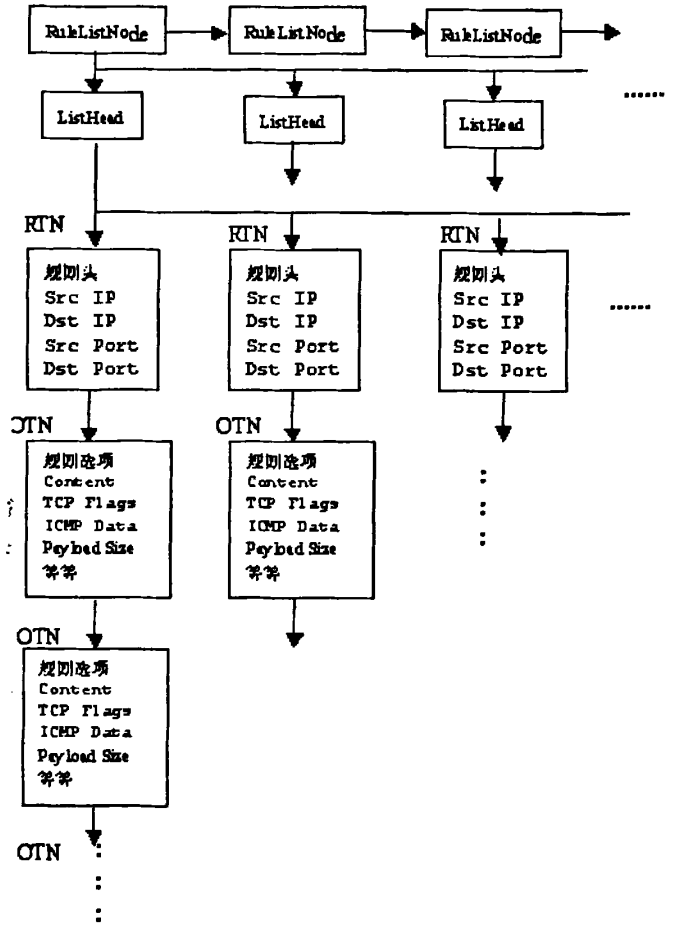

图2.4.4.1　规则的内存表示层次结构图

第一层为 RuleListNode 结构。RuleListNode 链表代表了系统所支持的规则类型（Alert、Pass、Log、Activation、Dynamic）。

第二层为 ListHead 结构。ListHead 中又按照协议类型（Ip、Tcp、Udp、Icmp）划分的 RuleTreeNode 构成的子规则链表。

第三层是最基础的数据结构 RuleTreeNodes（RTN，规则头节点）。

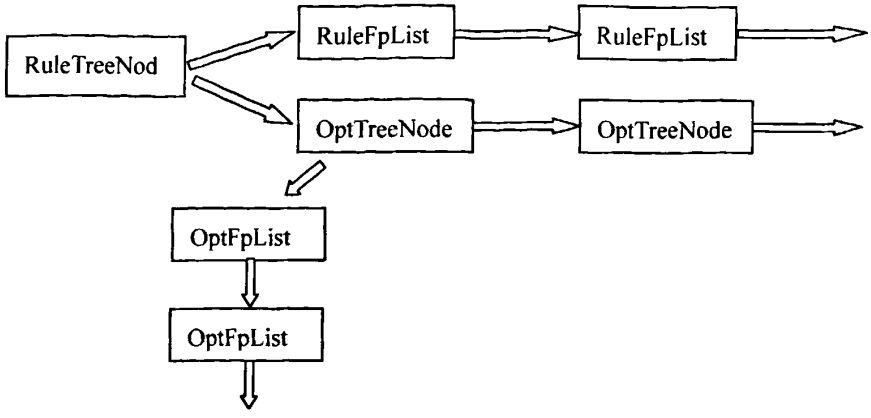

图2.4.4.2 RuleTreeNodes规则节点的内存表示结构图

每条规则包括规则头和规则选项两部分，规则头的匹配在RTN结构中完成，考虑到可能有很多规则，其规则头相同，而规则选项部分不同，则将它们放到同一个RTN下面的OTN链表，这样每个OTN都代表一条规则，且这些规则的规则头部分只用做一次匹配。

规则头节点结构中包含一个RuleFpList函数链表，这些函数都是用来处理规则头的信息的函数。如果规则头能够匹配，则进入下一层链表OptTreeNode进行规则选项匹配。

第四层也是最基础的数据结构 OptionTreeNodes（OTN，规则选项节点）。

规则选项节点结构中根据选项内容需要调用相应的处理函数，因此包含一个规则选项函数 OptFpList 函数链表。

每个 OTN 有一个规则选项处理函数 OptFpList 链表，该链表的每个节点存放一个选项匹配函数链表，该链表中放置的是处理该选项内容需要调用的函数。

2. 规则解析

获取\rules 文件夹中的规则文件，并根据 Snort 配置文件的规定，读入相应的规则，解析后，生成规则链表，然后再根据规则链表，按照 IP 地址、端口进行分类，并构造相应的快速规则链表。

关键的检测规则解析任务由 parser.c 文件中的函数 ParseRulesFile 负责完成，在 SnortMain 中调用。实质上该函数不仅是解析检测规则集合，而且对所有的系统配置规则都进行解析，包括预处理器、输出插件、配置命令等。因此 ParseRulesFile 是 Snort 系统的一个核心关键模块。但是，ParseRulesFile 只是一个过渡接口，其主要功能就是读取规则配置文件的每一行，并送到实际的规则解析模块 ParseRule 进行解析。ParseRule 的功能是解析所有的系统配置规则，包括插件配置、检测规则配置、变量定义、类型定义等。ParseRule 函数对配置文件的每一行（从 ParseRuleFile 函数传过来）进行解析。

从下面的函数中可以清晰地看出规则链的生成细节。

第一个函数——规则文件解析函数 ParseRulesFile，其功能就是逐行读取规则文件，并调用规则解析函数进行相应的处理。它包含三个参数：file=>规则文件名，inclevel=>进栈”include” 的规则文件级数，parse_rule_lines=>解析规则行数。返回值为空。

在函数中，打开规则文件，逐行读取文件内容，对文件内容中的一些特殊字符的处理，如注释符“#”、回车符等进行相应的处理；对于一条跨行书写的规则，也要进行多行判断处理。调用规则解析函数 ParseRule，进行规则解析，并检查规则的完整性。

第二个函数——规则解析函数 ParseRule。ParseRule 函数的基本算法流程如图 2.4.4.3 所示，源程序见 Snort 源文件目录中的 parser.c。

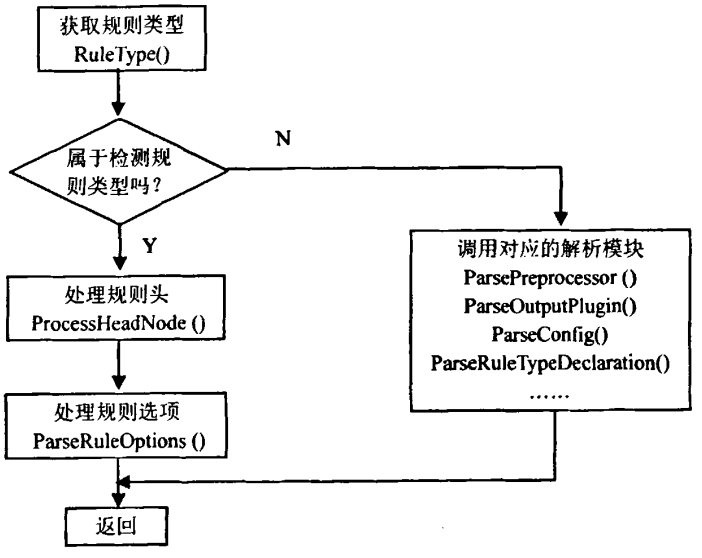

图2.4.4.3 规则解析函数的工作流程

ParseRule 函数的功能是处理单个规则并把它添加到规则链表中，包含四个参数：rule_file =>规则文件名，prule =>规则指针，inclevel=>进栈”include” 的规则文件级数，parse_rule_lines=>解析规则行数，返回值为空。

首先调用 mSplit 函数，对规则进行字段拆分，共 8 项，存放在 toks[]数组中，分别对应动作、协议、源 IP、源端口、攻击方向、目的 IP、目的端口和规则选项。规则解析函数的工作流程中的“属于规则检测类型”的判断就是依据 toks[0]的值，如果 toks[0]的值为 alert/log/pass/activate/dynamic，是由 include 语句引入，则进行对应规则解析处理。若为其它值，则分别调用相应的解析函数进行处理。

经过ParseRuleFile和ParseRule的处理后，系统内部就生成如图2.4.4.1的规则链表结构，接下去就是在这个基本的规则链表上，进一步划分构建新的数据结构，便于进行快速规则匹配。

第三个函数——规则头解析函数 ProcessHeadNode，它负责处理规则头信息，并根据需要把它添加到相应的规则链表中。函数包含三个参数：test_node => 由规则解析函数生成的数据，list => 列表头指针， protocol => 协议类型，函数返回值为空。

函数中首先选择合适的协议进行规则链表添加操作，如果表头为空，新建一个动作 Alert/Log/Pass/Activation/Dynamic 的表头节点并进行添加；如果表头不为空，即存在该动作链表，则检查是否有相同的规则头节点，若无，则新建一个节点，并把它添加到规则头节点链表末尾，并为新 RTN 节点初始化函数列表。

在规则头解析函数中，调用了为新RTN节点初始化函数链表函数 SetupRTNFuncList，该函数是规则和规则匹配引擎之间的桥梁。Snort在对规则的关键字做解析，并在内存中形成规则链表后，还没有和关键字的解释代码联系起来。SetupRTNFuncList正是实现规则链表关键字和规则匹配引擎关联的函数。

SetupRTNFuncList(RuleTreeNode *)为新RTN节点初始化函数列表函数中，还分别调用了AddRuleFuncToList函数（把相应的检测函数添加到规则头的RuleFpList链表中）；AddrToFunc函数（添加相应的地址到规则链表中）和PortToFunc函数（添加相应的端口到规则链表中）。

第四个函数——规则选项解析函数ParseRuleOptions。ParseRuleOptions函数负责处理选项部分，并把它添加到规则链表中合适的位置。它遍历括号内的规则选项，通过匹配规则选项关键字来调用相关的解析函数。该函数有三个 参数：rule => 规则字符串，rule_type =>列举规则类型 (alert, pass, log)，protocol =>协议类型，函数返回值为空。

首先判断OTN节点链表是否为空，不为空，则新建一个OTN节点，并添加到链表末端；为空，则新建一个链表。规则选项包含许多的关键字，分离不同的规则选项关键字，并存储到相应位置，并按照各关键字不同的处理方法，将处理函数加载到规则函数链表中。在规则选项解析函数中，调用AddOptFuncToList函数，将规则选项关键字和它的处理函数进行关联。至此，规则链表建立完成。

3. 快速规则匹配模块

Snort 2.0 以上版本重新构造了一套用于快速规则匹配的数据结构，快速规则匹配模块是 Snort 系统中最重要的模块，其设计的好坏直接影响到系统的性能，Snort 检测模块采用典型的特征匹配算法，并使用了新的多模式搜索匹

配引擎，大大提高了规则检测性能。

Snort 2.0 的快速规则检测模块，其工作流程主要分为以下三步[26]：

（1） 构造初始的规则链表结构，该步骤主要由 ParseRulesFile 和 ParseRule 两个函数完成。

（2） 构造用于快速规则匹配的新数据结构，这是在初始化各个插件和建立规则的三维链表后，在调用处理模块函数 InterfaceThread 前完成的。它是由 fpcreate.c 文件中的 fpCreateFastPacketDetection 函数完成的。

（3） 对当前数据包执行具体的快速规则匹配任务，其主要的接口函数为 fpEvalPacket。

前面已经对第一个步骤作了详细介绍，下面对这第二、第三个步骤的函数分别介绍。

第一个函数——构造快速规则匹配引擎 fpCreateFastPacketDetection

Snort 2.0系统中快速规则匹配引擎主要是通过有效地划分规则集合以及采用多模匹配思想实现的。系统采用的基本思想是通过规则中的目的端口和源端口值来划分类别，其要点如下：

（1）如果源端口值为特定值，目的端口为任意值(ANY)，则该规则加入到源端口值对应的子集合中；如果目的端口也为特定值，则该规则同时加入到目的端口对应的子集合中。

（2）如果目的端口为特定值，而源端口为特定值(ANY)，则该规则加入到目的端口对应的子集合中；如果源端口也为任意值(ANY)，则该规则同时加入到源端口对应的子集合中。

（3）如果目的端口和源端口都为任意值(ANY)，则该规则加入到通用子集合中。

（4）对于规则中端口值求反操作或者指定值范围的情况，等同于端口值为ANY情况加以处理。

此外，系统对于不含端口值的规则类型（ICMP 或 IP 协议类型），采用如下处理方法：

（1）对于ICMP协议规则，如果规则中指定了ICMP类型值，则目的端口值指定为ICMP类型值；否则，规定为任意值（ANY ）。

（2）对于IP协议类型，如果规则中指定了IP高层类型值，则目的端口值指定为IP高层协议类型值；否则，规定为任意值（ANY）。

（3）两者规则划分时，源端口值都指定为任意值(ANY)。

在构造快速规则匹配引擎时，系统根据协议类型重新构造了快速匹配的规则链表，定义了4个PORT_RULE_MAP类型的全局变量，变量定义在pcrm.h文件中，它

们是：

```
Static PORT_RULE_MAP *prmTcpRTNX=NULL;
Static PORT_RULE_MAP *prmUdpRTNX=NULL;
Static PORT_RULE_MAP *prmIpRTNX=NULL;
Static PORT_RULE_MAP *prmIcmpRTNX=NULL;
```

PORT_RULE_MAP这种类型的结构为UDP/TCP协议划分了源、目的端口，为IP协议划分了目的端口。PORT_RULE_MAP的数据结构如下：

```
typedef struct {
   int         prmNumDstRules;
   int         prmNumSrcRules;
   int         prmNumGenericRules;
   int         prmNumDstGroups;
   int         prmNumSrcGroups;
   PORT_GROUP *prmSrcPort[MAX_PORTS];
   PORT_GROUP *prmDstPort[MAX_PORTS];
   PORT_GROUP *prmGeneric;
} PORT_RULE_MAP ;
```

在ORT_RULE_MAP的数据结构中，定义了2个PORT_GROUP类型的数组和1个PORT_GROUP类型的链表指针。前2个数组代表所有特定端口值（目的和源端口）对应的规则子集合的总和，而后面的链表指针代表通用规则子集合。

PORT_RULE_MAP数据结构中所涉及的数据结构类型PORT_GROUP，其负责存放根据特定端口值划分的规则子集合。在PORT_GROUP结构中，定义了三组相同结构类型的规则节点链表，分别表示包含content内容匹配选项的规则、包含uricontent匹配选项的规则和没有内容匹配选项的规则。

在RULE_NODE结构类型定义中又使用了OTNX结构类型，OTNX结构类型中包含了分别指向特定规则所对应的链表头节点和选项节点的指针值，其定义在fpcreate.h文件中，具体结构如下：

```
typedef struct _otnx_{
   OptTreeNode   * otn;
   RuleTreeNode  * rtn;
   unsigned int    content_length;
} OTNX;
```

根据划分标准而得到的各个规则子集合中，实际包含的都是OTNX结构类型的指针值，所指向的是初始规则链表结构中的对应节点。如此将快速规则匹配引擎

中的新构建的数据结构与原始的规则链表RTN、OTN数据结构联系起来。其中，OTNX数据结构是关键的桥梁。

划分规则子集合的函数fpCreateFastPacketDetection（）进行构建操作时，利用上述四个变量来存放规则子集合的地址。其主流程算法如图2.4.4.4所示，源程序见Snort源文件目录中的fpcreate.c。

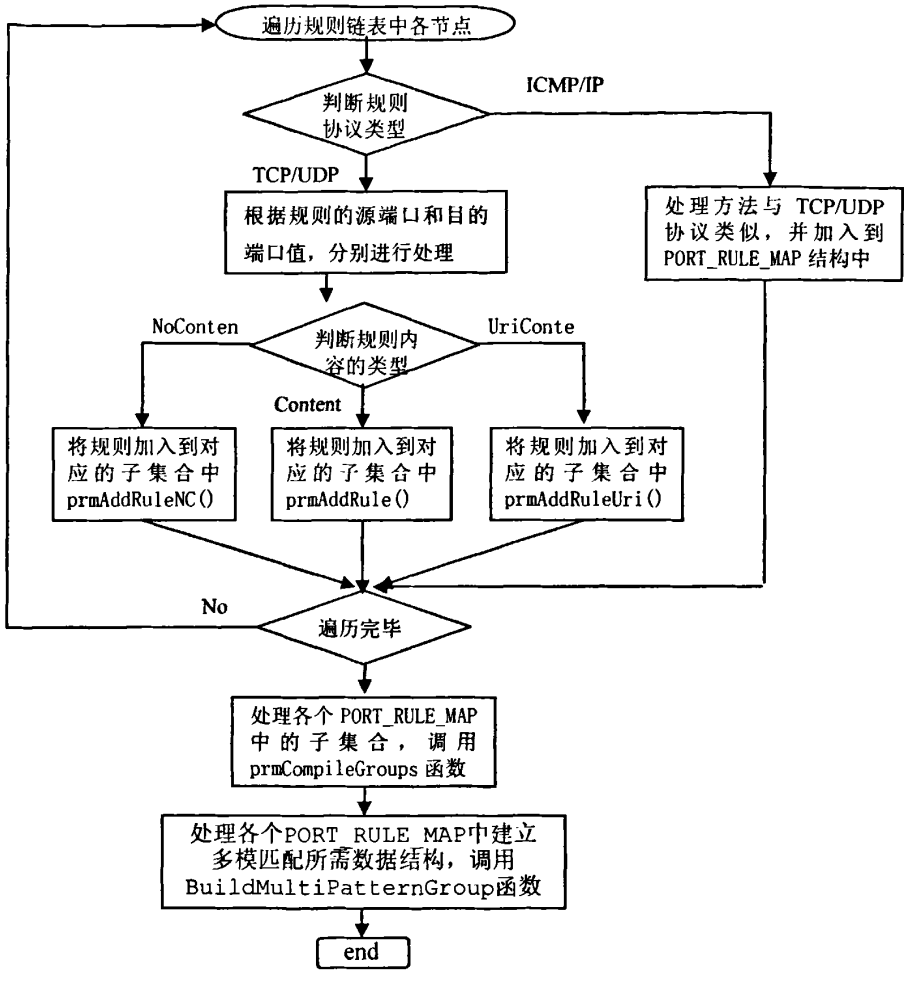

图2.4.4.4　快速规则匹配引擎的构建流程

fpCreateFastPacketDetection函数遍历RuleListNode->ListHead->RTN/OTN类型的链表（如：rule->RuleList->TcpList），在此每一条规则根据其内容(Content/UriContent/)被分类，内容信息存储在OTN中：在OTN中有一项指针数组ds_list，指向不同的根据规则设置的数据结构。根据每一条规则的ds_list内容类型分别调用prmAddRule/prmAddRuleUri/prmAddRuleNC函数。fpCreateFastPacketDetection函数根据各条规则中给出的源端口和目的端口把规则顺序加入到合适的

PORT_RULE_MAP -> PROT_GROUP -> RULE_NODE链表中。这样做的目的是为了能够快速和捕获包进行特征匹配。

实际执行规则集合划分标准的功能模块为prmAddRule()、prmAddRuleUri()和prmAddRuleNC()。同时，这三个功能模块还将根据当前规则链表选项节点是否包含内容匹配选项及类型，将当前规则节点加入到对应PORT_GROUP结构中三组同类型的规则节点链表中。

以prmAddRule()为例，该函数根据源端口和目的端口值，选定当前PORT_RULE_MAP中对应的PORT_GROUP结构，然后根据调用prmxAddPortRule()来具体执行在指定PORT_GROUP结构中的对应规则节点链表中加入当前的OTNX结构变量。对其代码进行解析如下：

```
int prmAddRule( PORT_RULE_MAP * p, int dport, int sport, RULE_PTR
rd )
{
   /* 若根据目的端口（0 -> 65535）来汇聚信息 */
   if( dport != ANYPORT && dport < MAX_PORTS )
   prmxAddPortRule( p->prmDstPort[ dport ], rd );
   /*若根据源端口（0 -> 65535）来汇聚信息 */
   if( sport != ANYPORT && sport < MAX_PORTS)
   prmxAddPortRule( p->prmSrcPort[ sport ], rd );
   /*指本规则可应用于任何值 */
   if( sport == ANYPORT && dport == ANYPORT)      //ANYPORT==-1
   prmxAddPortRule( p->prmGeneric, rd );
}
```

继续调用prmxAddPortRule函数最终把RULE_NODE节点加入到PORT_GROUP形成规则内容链表：

```
static int prmxAddPortRule( PORT_GROUP *p, RULE_PTR rd )
{
   p->pgTail->rnNext = (RULE_NODE*)malloc( sizeof(RULE_NODE) );
   p->pgTail= p->pgTail->rnNext;
   p->pgTail->rnNext = 0;
   p->pgTail->rnRuleData = rd;
}
```

注意现在存在两个链表：一个是RuleListNode->ListHead->RTN/OTN/OutputFuncNode链表，表头是parser.c文件

中全局变量RuleLists；另一个是PORT_RULE_MAP -> PROT_GROUP -> RULE_NODE链表，表头是fpcreate.c文件中全局数据结构指针prmTcpRTNX/prmUdpRTNX/prmIcmpRTNX/prmIpRTNX。

通过第一个链表中 OTN 的规则内容建立第二个链表[27]。

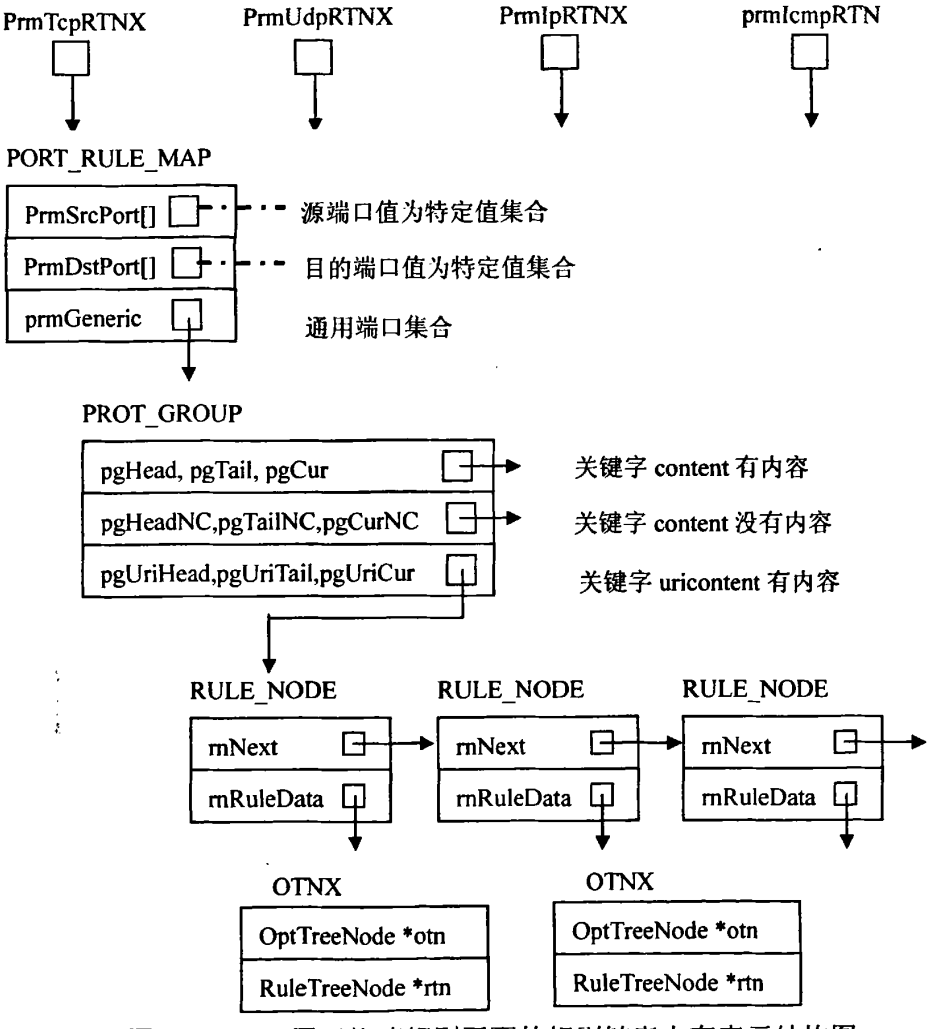

图 2.4.4.5　用于快速规则匹配的规则链表内存表示结构图

函数fpCreateFastPacketDetection（）在完成规则链表的遍历操作后，即完成了各个规则子集合的初步划分，所有的规则节点都加入到如下三种类型的子集合中：

（1）对应特定源端口值的 PORT_GROUP 结构的规则链表。

（2）对应特定目的端口值的 PORT_GROUP 结构的规则链表。

（3）通用规则节点链表。

对于通用规则节点链表而言，在构建快速规则匹配引擎中，它仅是作为一个过渡性的数据结构，对于未来的检测任务而言，每个数据包都有特定的源/目的

端口值，为了达成根据端口值进行快速规则匹配的任务，函数fpCreateFastPacketDetection在完成遍历操作后，对每个PORT_RULE_MAP结构中的通用规则链表进行了进一步的处理，调用prmCompileGroups函数，该函数的主要功能就是将通用规则链表中的各个规则节点加入到另外对应于特定端口值的两个规则链表中，即在PORT_RULE_MAP结构中的PORT_GROUP数组中每个非空元素中加入。

最后，为了适应多模式搜索引擎算法的需要fpCreateFastPacketDetection函数调用BuildMultiPatternGroup函数，在每个PORT_ GROUP结构中构建多模式搜索引擎所需要的数据结构。

4. 规则检测

Snort 系统执行数据包处理函数 ProcessPacket，该函数调用解码和预处理函数处理数据包，在预处理函数 Preprocess 中，处理完预处理模块后，调用 Detect 函数对捕获的 packet 结构类型参数的数据包内容进行特征规则匹配。该函数调用 fpEvalPacket 函数，首先检查高层的协议，如果是 TCP、UDP 或 ICMP 协议的数据包，则使用各协议相对应的查找函数 fpEvalHeaderTcp(p)、fpEvalHeaderUdp(p)、fpEvalHeaderIcmp(p)，若不能归为这三种协议的就算作 Ip 协议，使用 IP 协议对应的查找函数 fpEvalHeaderIp(p, ip_proto)。以 Tcp 协议为例，调用 fpEvalHeaderTcp 函数。

下面以 fpEvalHeaderTcp 函数源码为例解析：

```
static INLINE int fpEvalHeaderTcp(Packet *p)
{ /*根据给定参数的源端口和目的端口决定应根据哪个 PORT_RULE_MAP->
    PORT_GROUP 结构进行匹配 */
   prmFindRuleGroupTcp(p->dp, p->sp, &src, &dst, &gen);
   /*遍历 Tcp 类型的 PORT_RULE_MAP->PORT_GROUP 链表进行 http_decode
模式特征匹配*/
   fpEvalHeaderSW(dst, p, 1);
}
```

函数fpEvalHeaderTcp首先调用prmFindRuleGroupTcp函数获取当前可用的规则子集合。然后根据返回值，分别以不同的规则子集合指针参数调用快速规则匹配函数fpEvalHeaderSW，执行具体的规则匹配任务。这就体现了规则集合划分的特点及按照规则子集合进行匹配的优势。

fpEvalHeaderSW函数遍历PORT_RULE_MAP->PORT_GROUP链表，进行uri-conetnt/content/non-conetnt匹配，下面对其进行解析。

```
int fpEvalHeaderSW(PORT_GROUP *port_group, Packet *p, int
```

```
check_ports)
    {  /*循环 HttpUri 类型全局指针数组 UriBufs[URI_COUNT] */
    for( i=0; i<p->uri_count; i++)
    {  /* 以下分别进行 uri-conetnt/content/non-conetnt 匹配，仅以
    content 为例*/
     stat = mpseSearch ( so, p->data, p->dsize, otnx_match, &omd );
     fpLogEvent(otnx->rtn, otnx->otn, p);
     }
     }
```

继续分析上面调用的函数：

（1）mpseSearch 函数：

SNORT 提供三种完全不同的模式匹配算法：Aho-Corasick/Wu-Manber/Boyer-Moore，默认使用第三种。注意 mpseSearch 函数中有两个参数分别是： 所要匹配的数据包指针(Packet*)和先前为了快速匹配特征串而建立的对应的结构指针（PORT_RULE_MAP->PORT_GROUP*），后面所有调用的函数参数都是从这两个参数中得到。

mpseSearch 函数调用 mwmSearch 函数，mwmSearch 函数继续调用 hbm_match 函数，这个函数即是用 Boyer-Moore-Horspool 模式匹配算法进行特征匹配。具体算法实现请参照 msting.c 文件，在下一章详细叙述。

（2）fpLogEvent 函数：

若匹配成功，则调用本函数记录或报警。该函数的参数分别是：匹配的 RTN/ONT 指针和数据包指针(Packet*)，具体实现如下：

```
switch(rtn->type)
{
case RULE_PASS: PassAction();
case RULE_ACTIVATE: ActivateAction(p, otn, &otn->event_data);
case RULE_ALERT: AlertAction(p, otn, &otn->event_data);
case RULE_DYNAMIC: DynamicAction(p, otn, &otn->event_data);
case RULE_LOG: LogAction(p, otn, &otn->event_data);
}
```

若是应用 pass 规则动作，则仅设置通过包的数目加一；

若是 dynamic/log 规则动作，则调用 detect.c 文件中的 CallLogFuncs 函数进行记录；

若是activate/alert规则动作，则调用detect.c文件中的CallAlertFuncs

函数进行报警。

5. 规则处理函数

Snort系统提供的处理模块被称为检测引擎插件，它们主要用来对数据包进行检查，以发现入侵。处理模块的源代码都集中在detection-plugins子文件夹中以sp_开头的文件，Snort-2.7.0.1中共有32种对应的检测文件。这些模块会检查数据包的各个方面，包括数据包的大小、协议类型等。根据用户在规则文件中的设定，可能会对一个数据包多次调用同一个处理模块，参数会有所不同。例如：sp_dsize_check.c模块检查数据包的大小、sp_ip_same_check.c模块用于检查IP报文的源地址与目的地址是否相同、sp_pattern_match.c模块负责检查数据包中是否包含各规则指定的特征字符串等。

规则处理模块在初始化模块InitPlugIns()函数中已经和规则选项关键字关联，并在规则解析中插入到OptFpList链表中。

规则处理模块的差别很大，有的比较复杂，有的相对简单。下面仅对sp_pattern_match进行解释。

sp_pattern_match模块是Snort处理模块中的史前巨兽，也是功能最强大的部分，它负责检查数据包、搜索用户在规则文件中指定的特征字符串等。下一章将具体分析此模块的各种模式匹配算法。

2.4.5 Snort 系统流程总结

Snort功能实现可以分成两大块：一是Snort系统的初始化，二是在Snort系统功能框架建立起来后，抓包并进行模式匹配。

1. 初始化分两步：第一步把所有的三类插件（output/preprocess/plugins）串成三个链表
（OutputKeywordList/PreprocessKeywordList/KeywordXlateList）以供下一步调用；第二步是读取配置文件，并从上一步中插件链表中挑出配置文件中相应的要用到的插件建立规则链表
RuleListNode->ListHead->RTN/OTN/OutputFuncNode，这其中的RTN/OTN链表构成Snort的有特色的三维链表，供模式匹配用。

2. 模式匹配。分两步：第一步，在模式匹配初始化时另外建立一个PORT_RULE_MAP -> PROT_GROUP -> RULE_NODE链表以供快速匹配用；第二步循环抓包并用配置好的模式匹配算法实现。

2.5 本章小结

本章全面介绍的 Snort 系统，介绍其特点，详细介绍了 Snort 的规则和总体结构，并根据 Snort 源代码对 Snort 流程进行了详细的分析。

3 字符串模式匹配算法研究

3.1 字符串模式匹配算法概述

串匹配是计算机研究领域中最经典、研究最广泛的问题之一，是许多应用系统的核心技术之一。在现实生活中，串匹配技术的应用十分广泛，其主要应用领域包括：入侵检测、病毒检测、信息检索、计算生物学、金融监测等等。在许多应用系统中，特征串的匹配所占的时间比重相当大，串匹配算法的速度很大程度上影响着整个系统的性能。

对于串匹配算法的研究，已经有三十多年的历史，计算机科学家们提出了很多优秀的算法，并且还在不断地探索之中。在入侵检测领域中，采用特征匹配的误用检测引擎的主要检测方法是通过将网络数据包同已有规则进行匹配以甄别攻击，所以采用合适的串匹配算法对于误用型网络入侵检测系统的效率有着至关重要的影响。

字符串模式匹配的定义如下，一个文本串 text，它的长度为 n，一个用于进行匹配的字符串称为模式 pattern，它的长度为 m，一般 m<n，字符串的字符取自扩展的 ASCII 码，如果在 text 中发现一个和 pattern 完全匹配的字符串，称为字符串模式匹配；如果在 text 中没有发现 pattern 字符串，成为字符串模式失配。

评价字符串模式匹配算法主要从两方面进行，一使算法时间复杂度，二是算法的空间复杂度。设计出复杂度尽可能低的算法是算法设计追求的一个重要目标；当给定的问题有多种算法时，选择其中复杂度最低者，是选用算法所遵循的一个重要准则。

3.2 模式匹配算法的分类

从匹配的精确度上来划分，模式匹配算法可以分为精确匹配算法和近似匹配算法。

精确匹配算法在匹配过程中不允许存在误差匹配，即模式串中的所有字符必须和文本串中的字符一一对应，才算是匹配成功。而近似匹配算法允许模式和文本字符串之间存在某些细微的差异。近似匹配在信息检索和计算生物学领域有着广泛应用，字符串中包含通配符、空位和正则表达式。

从算法一次能够匹配的模式数量上来划分，模式匹配算法分为单模式匹配算法和多模式匹配算法。

单模式匹配算法每次只能在文本串 T 中查找出一个特定的子串 P。如果 T 中找到等于 P 的子串，则称匹配成功；否则匹配失败。

多模式匹配算法可以在文本 T 中一次查找模式集{P}的所有模式，找出是否有模式集中的一个或多个模式的出现。

在实际的网络入侵检测系统中，为了保证检测的准确性，降低入侵检测系统的误报率，入侵检测系统一般都采用精确的模式匹配算法。同时随着网络技术的快速发展，入侵规则库日益庞大，运用的模式匹配算法也由单模式匹配算法向多模式匹配算法过渡。

根据在文本中的搜索方式不同，可分为基于前缀搜索方法、基于后缀搜索方法和基于子串搜索方法[33]。在这三种搜索中，都定义了一个搜索窗口，其长度与最短模式串长度相等，搜索窗口沿文本从左向右移动或从右向左移动，串匹配在搜索窗口中进行。

基于前缀的搜索：在搜索窗口内从左向右(沿着文本的正向)逐个读入文本字符，搜索窗口中文本和模式串的最长公共前缀。著名的算法有KMP（Knuth-Morris-Pratt）算法、Aho Corasick算法、Shift-And 或 Shift-Or算法等。

基于后缀的搜索：在搜索窗口内从右向左(沿着文本的反向)逐个读入文本字符，搜索窗口中文本和模式串的最长公共后缀。使用该方法可以跳过一些文本字符，著名的算法Boyer-Moore算法、Horspool算法、Quick Search算法和多模式串的Commentz-Walter算法、Wu-Manber算法等。

基于子串的搜索：在搜索窗口内逐个读入文本字符，搜索窗口中文本和模式串的最长子串，可以看作是后缀搜索和前缀搜索的结合。使用该方法可以跳过一些文本字符，但是需要识别模式串的所有子串，比较复杂。著名的算法有BDM（Backward DAWG Matching）算法和BOM（Backward Oracle Matching）算法，以及相应的多模式串匹配算法SBDM（Set Backward DAWG Matching）算法和SBOM（Set Backward Oracle Matching）算法。

3.3 Snort 检测引擎的模式匹配算法

Snort 系统提供了可选择的搜索匹配算法，Boyer-Moore 算法/Aho-Corasick 算法/Wu-Manber 算法。Snort 2.0 版本之前的系统检测引擎主要采用 BM 模式匹配算法，实现此算法的模块为 mstring.c 文件。Snort 2.0 以后版本加入了基于 Aho-Corasick 多模式搜索引擎来加快系统的匹配速度，实现该算法的模块为 acsmx2.c 文件。下面对几种比较常用的快速算法分别进行分析。

3.3.1 Boyer-Moore 算法

Boyer Moore 算法[13][34][33]是一种基于后缀的匹配算法。搜索窗口从文本的左端向右移动，匹配从窗口右端开始向左逆向匹配，在得到部分匹配时充分地利用部分匹配所隐含的信息，帮助跳过不必要的比较，提高算法效率。Boyer Moore 算法通过计算三个移动函数来决定下一次匹配动作模式的开始位置，d1，

d2 和 d3。下面对这三种情况分别进行介绍。

(1) 良好后缀移动情况一

假设模式的后缀 u 和窗口中的文本后缀 u 匹配，P[i+1: m-1]=T[i+j+1: j+m-1]=u 而且不匹配发生在 P[i]处，P[i] ≠ T[i+j]。此时在模式中还存在其它 u，将模式中最右出现的 u 移动到与文本 u 相匹配的位置。如图 3.3.1.1。

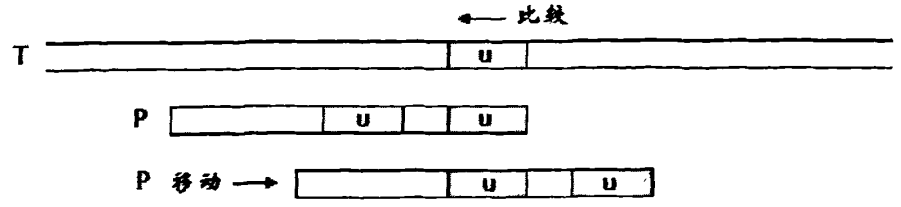

图 3.3.1.1　匹配的后缀出现在模式串的其他地方

(2) 良好后缀移动情况二

假设模式的后缀 u 和窗口中的文本后缀 u 匹配，P[i+1: m-1]=T[i+j+1: j+m-1]=u 而且不匹配发生在 P[i]处，P[i] ≠ T[i+j]。此时在模式中不存在其它 u，但是存在后缀 u 的后缀 v，将模式中的 v 移动到与文本 v 相匹配的位置。如图 3.3.1.2。

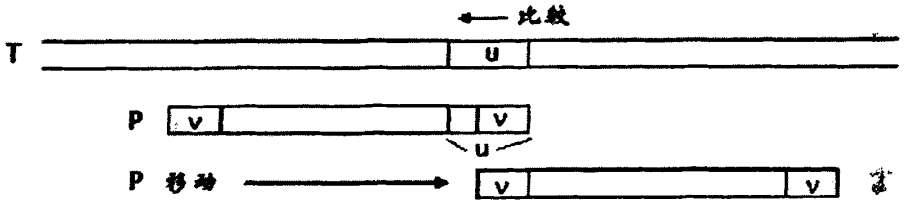

图 3.3.1.2　匹配的后缀的后缀出现在模式串的其他地方

(3) 坏字符移动情况一

假设模式的后缀 u 和窗口中的文本后缀 u 匹配，P[i+1: m-1]=T[i+j+1: j+m-1]=u 而且不匹配发生在 P[i]处，P[i] ≠ T[i+j]，T[i+j] =b。如果按照第一种情况进行窗口移动，移动后对应的字符不是 b，显然这次匹配不是必要的。若模式中存在字符 b，将模式移动到字符 b 对齐的位置。如图 3.3.1.3。

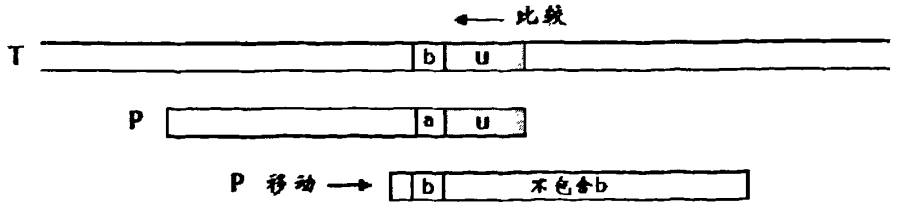

图 3.3.1.3　模式移动到下一次字符 b 出现的地方

(4) 坏字符移动情况二

若模式中不存在字符 b，将模式移动到文本字符 b 后的位置。如图 3.3.1.4。

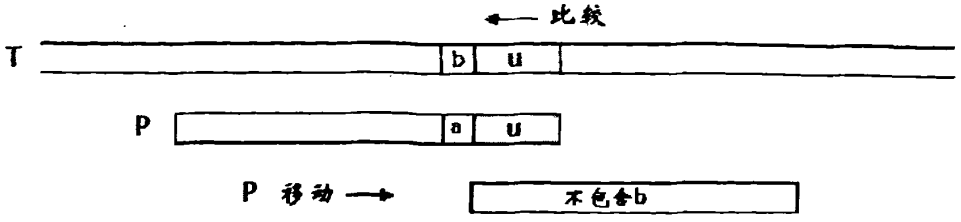

图 3.3.1.4　模式移动到文本字符 b 后的位置

Boyer Moore 算法的真正移动距离则是由坏字符转移和好后缀转移中移动距离最大的决定。

例如：我们要在"gcatcgcagagagtatacagtacg"查找"gcagagag"，分别计算 bmBc[]和 bmGs[]，结果见表 3.3.1.1 和表 3.3.1.2[35]。

表 3.3.1.1　bmBc 函数值

i	a	c	g	其他字符
BmBc[i]	1	6	2	8

表 3.3.1.2　qsBc 函数值

位置	0	1	2	3	4	5	6	7
BmGs	7	7	7	2	7	4	7	1

匹配过程：

gcatcgcagagagtatacagtacg

.....g　　根据失配文本字符 a 移动，bmBc[a]=1

 .....gAG　　根据失配文本字符 c 移动，bmBc[c]-2=4

　　GCAGAGAG　　匹配成功，根据后缀移动七位(bmGs[0])

　　　　.....gAG　　根据失配文本字符 c 移动，bmBc[c]-2=4

　　　　　.....aG　　根据失配文本字符 c 移动，bmBc[c]-2=4

至此，匹配完成。

Snort 系统中的 BM 算法在 mstring.c 文件中定义。

3.3.2 Boyer-Moore-Horspool（BMH）算法

BMH算法[14][34][33]是BM算法的简化实现，在搜索窗口中发生不匹配时，并不关心是文本中哪个字符造成了失配，而是简单地使用窗口最右端的文本字符来决定右移量，即只使用坏字符移动。预处理阶段的时间复杂度是O(m+n)，空间复杂度为O(n)，搜索阶段的时间复杂度为O(mn)。

开始搜索时，文本与搜索窗口左对齐，窗口从左向右移动。在匹配窗口中，匹配从右向左进行，一旦失配，根据窗口最右端文本字符来移动窗口。

bmhBc[c]计算如下：

bmhBc[c] = min{i|1≤i <m-1, x[m-1-i]=c} ，c 出现在模式中

bmhBc*[c] = m, c 不在模式中*

例如：我们要在"gcatcgcagagagatacagtacg"查找"gcagagag"，计算 bmBc []，结果见表 3.3.2.1。

表 3.3.2.1　　BMH 算法的坏字符转移表

字符	a	c	g	其他字符
模式移动距离	1	6	2	8

gcatcgcagagagtatacagtacg

gcagagag　　窗口右端本文 a 决定右移一位，bmBc[a]=1

 gcagagag　　窗口右端本文 g 决定右移二位，bmBc[g]=2

  gcagagag　　窗口右端本文 g 决定右移二位，bmBc[g]=2

   gcagagag　　窗口右端本文 g 决定右移二位，bmBc[g]=2

    gcagagag　　窗口右端本文 a 决定右移二位，bmBc[a]=1

     gcagagag　　窗口右端本文 t 决定右移八位，bmBc[t]=8

         gcagagag　　窗口右端本文 g 决定右移二位，bmBc[g]=2

至此，匹配结束。

3.3.3 Quick Search (QS)算法

Quick Search (QS) 算法[15][34][33]也是基于 BM 算法（Boyer-Moore）思想的一种改进算法，对于不匹配的情况只进行坏字符机制处理，而且处理方式改变为，移动距离决定于发生不匹配后，紧靠窗口右端的下一个文本字符。该算法较 BM 算法有了很大的改进，其匹配速度快于 BM 算法。QS 算法与 BMH 算法不同的是它考虑的是紧靠窗口右端的下一个正文字符，跳跃的最大值比 BMH 的大，但是在某些情况下并不比 BMH 快。

开始搜索时，文本与搜索窗口左对齐，窗口从左向右移动。在匹配窗口中，匹配也是从左向右进行，一旦失配，根据窗口后文本的下一个字符来移动窗口。

qsBc[c] = min{i+1|0≤i<m, x[m-1-i]=c} ，c 出现在模式中

qsBc[c] = m+1，c 不在模式中

例如：我们要在"gcatcgcagagagtatacagtacg"查找"gcagagag"，计算 qsBc []。

表 3.3.3.1　　QS 算法的坏字符转移表

字符	a	c	g	其他字符
模式移动距离	2	7	1	9

gcatcgcagagagtatacagtacg

GCAg....　　根据文本 qsBc[g]=1 右移一位

g.....　　根据文本 qsBc[a]=2 右移二位

g.....　　根据文本 qsBc[a]=2 右移二位

GCAGAGAG　　匹配成功。根据文本 qsBc[t]=9 右移九位，

g.....　　根据文本 qsBc[c]=7 右移七位

至此，匹配结束。

3.3.4 Aho- Corasick（AC）算法

3.3.4.1 多模匹配算法的引入

字符串的多模式匹配问题是要在给定的文本 T 中找出模式集合 P(1)……P(N)中所有出现的模式。一种最直接的方法是将每一个模式都与文本 T 进行一次匹配，但是这样的话，需要对文本T 进行N 次扫描和匹配。如果模式集合的许多模式中含有相同的子串，如比较有代表性的如 Snort 的 WEB-COLDFUSION 规则，共 35 项，其中有 15 项的 uricontent 具有相同的前缀“/cfdocs”，如下：

uricontent:"/cfdocs/cfmlsyntaxcheck.cfm"

uricontent:"/cfdocs/exampleapp/"

uricontent:"/cfdocs/exampleapp/email/application.cfm"

uricontent:"/cfdocs/exampleapp/email/getfile.cfm"

uricontent:"/cfdocs/exampleapp/publish/admin/addcontent.cfm"

uricontent:"/cfdocs/exampleapp/publish/admin/application.cfm"

uricontent:"/cfdocs/examples/cvbeans/beaninfo.cfm"

uricontent:"/cfdocs/examples/mainframeset.cfm"

uricontent:"/cfdocs/examples/parks/detail.cfm"

uricontent:"/cfdocs/expeval/"

uricontent:"/cfdocs/expeval/displayopenedfile.cfm"

uricontent:"/cfdocs/expeval/exprcalc.cfm"

uricontent:"/cfdocs/snippets/"

uricontent:"/cfdocs/snippets/evaluate.cfm"

uricontent:"/cfdocs/snippets/fileexists.cfm"

uricontent:"/cfdocs/snippets/gettempdirectory.cfm"

如果选择前面介绍的单模匹配，需要对这 15 条规则（即 15 个模式串）进行 15 次匹配，从观察我们可以知道，只要在 uricontent 中没有查找到“/cfdocs”

前缀，这 15 项规则内容就匹配不成功，而且在 Snort 规则中，具有相同前缀的规则比较普遍，寻找一种更加高效的匹配算法势在必行。如果只对文本 Text 进行一次扫描和匹配就完成对所有模式的搜索过程，而不论模式集合的数量有多大，这就是多模式匹配。目前常用的字符串多模匹配算法有 Aho- Corasick（AC）算法、Wu 和 Manber 的算法、SBOM 算法。本文只介绍 AC 算法。

3.3.4.2　AC 算法简介

1975 年，Aho 和 Corasick 提出了著名的基于有限状态自动机的 AC 算法[15]，这种算法采一种有限自动机命名为 Aho-Corasick 自动机，把所有的模式串构成一个集合，可以一次查找多个模式。这种算法被引入了入侵检测领域，成为一种经典的多模式匹配算法，后来的一些算法都是在它的基础上改进的。

Aho-Corasick 状态机是多模式搜索算法之一，该算法 1975 年产生于贝尔实验室[36]，最早被使用在图书馆的书目查询程序中，取得了很好的效果。有限状态机是系统所有可能状态的表示，以及该系统可接受的状态转换的信息。有限状态机的处理动作从最初状态开始，然后接受一个输入事件，根据输入事件从当前状态移动到下一个适当的状态。可以把状态机看作一个矩阵，行代表状态，列代表事件。根据输入的事件，可能要处理的动作以及进入或退出状态时的信息，矩阵元素提供了下一步要移动的适当状态。举例说明状态转换过程，如果当前状态是状态 10，并且下一输入事件为事件 6，则在矩阵中行为 10、列为 6 的元素值指出了从当前状态要进入的下一状态[37]。

Aho-Corasick 算法的基本思想是:在预处理阶段，有限自动机算法建立三个函数，转向函数 goto，失效函数 failure 和输出函数 Output。由此构造一个树型有限自动机。这样模式匹配的处理过程就变成了状态转换的处理过程，由“start ”状态开始。在搜索查找阶段，通过这三个函数的交叉使用扫描文本，定位出模式在文本中的所有出现位置。

构建转向、失效和输出函数包含两个部分，第一部分确定状态和转向函数，第二部分计算失效函数。输出函数的计算则是穿插在第一部分和第二部分中完成。

1、转向函数 G(s, t)。构建转向函数，需要建立一个状态转换图。开始时，状态转换图只包含一个状态 0。然后，通过添加一条从起始状态出发的路径的方式，依次向图中输入每个模式的字符 p[0]、p[1]……。新的顶点和边被加入到图表中，以致于产生了一条能拼写出模式 p 的路径。模式关 p 会被添加到这条路径的终止状态的输出函数中。当然只有必要时才会在图表中增加新的边[38]。

例如，对关键字集{spam, is, stop}建立转向函数和输出函数

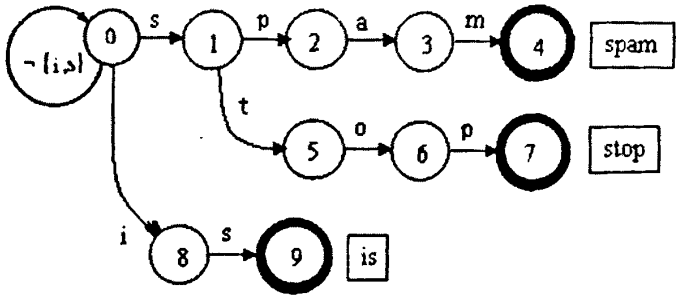

图 3.3.4.1　转向函数和输出函数

首先建成一棵带根的树，为了完成转向函数的构建，对除了 i 和 s 外的其他每个字符，都增加一个从状态 0 到状态 0 的循环。这样，就得到了如图 3.3.4.1 所示的状态转换图，这个图就代表转向函数和输出函数。

2、失效函数 F(s)。F(s)表示在状态 s 时，如果下一个输入 t 不在任何一个转向函数表示的箭头上，即一个输入字符导致了不匹配，那么自动机的状态将转到 F(s)，失效函数被用来选择一个状态，由这个状态重新开始匹配。即如果 G(s, t)不存在，则转到 G(F(s), t)状态。

失效函数是根据转向函数建立的。F(s)的构造方法为：令所有第一层状态的失效函数为 0，即 F(1)=F(3)=0；对于非第一层状态 s，若其父状态为 r（即存在字符 t，使 G(r, t)=s，则 F(s)=G(F(s*), t)，其中状态 s*为追溯 s 的祖先状态得到的第一个使 G(F(s*), t)存在的状态。[36]

如上例：　G(1, p)=2，F(2)=G(F(1), p)= G(0, p)=0；

G(2, a)=3，F(3)=G(F(2), a)= G(0, a)=0；

G(3, m)=4，F(4)=G(F(3), m)= G(0, m)=0；

G(8, s)=9，F(9)=G(F(8), s) =G(0, s)=1。

F(s)的值如表 3.3.4.1 所示。

表 3.3.4.1　　失效函数 failure

s	1	2	3	4	5	6	7	8	9
F(s)	0	0	0	0	0	0	0	0	1

三个函数建立完毕，即得到如图 3.3.4.2 所示的 Aho-Corasick 自动机。

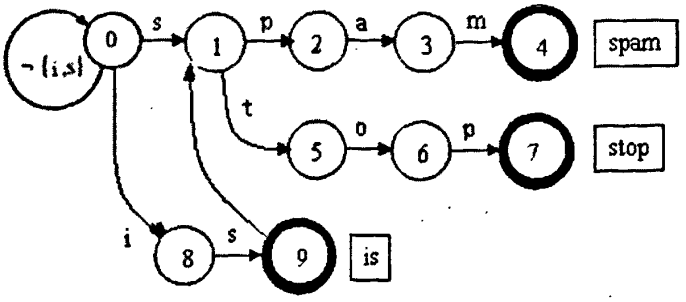

图 3.3.4.2　Aho-Corasick 自动机

利用已构成的有限自动机，在文本 T 为“antispam”中进行多个模式串的查找的过程如下：

从有限自动机的 0 状态出发，从目标字符串的第一个字符开始正向逐个取出的字符 t，并按转向函数 G(s, t)进入下一状态，如果 G(s, t)=fail，则按 G(F(s), t)进入下一状态。

当输出函数 P(s)不为空时，输出 P(s)。

对于上面的例子，具体的匹配的过程见图 3.3.4.3 所示。

当前状态	0	0	0	0	8	9（即 1）	2	3
输入字符	a	n	t	i	s	p	a	m
输入后的状态	0	0	0	8	9	2	3	4
输出					{is}			{spam}

图 3.3.4.3　　AC 算法模式匹配过程

当状态机 M 在状态 8 的时候并且当前的输入是 s，因为 G(8, s)=9，自动机进入状态 9，表明已经在状态 9 处发现关键字“is”。在状态 9，当输入字符是 p 时，因为 G(9, p)=fail，所以自动机自动进入状态 F(9)=1。

3.3.4.3 Snort 系统中 AC 算法描述

Aho-Corasick 算法在 Snort 源文件\sfutil\acsmx.c(h)文件中定义。其流程包括预处理阶段和搜索阶段。在预处理阶段构建 Aho-Corasick 自动机，建立转向函数 goto，失效函数 failure 和输出函数 Output。

Aho-Corasick($P=\{P_1, P_2, \cdots\cdots, Pq\}$, $T = t_1t_2\cdots\cdots t_n$)

1　Preprocessing

2　　*AC* ← *Build_trie* ($P=\{P_1, P_2, \cdots\cdots, Pq\}$)

3　　*AC* ← *computation_failure*(*P*)

4　Searching

5　　*Current* ← *root*

```
6  while i∈1…n  i<n do
7      if Current ← g(Current, text[i]) ≠ fail
8          IF isElement(Current, final)
9              RETURN TRUE
10         ENDIF
11     ENDIF
12 END while
```

```
build_trie (P={P1, P2, ……, Pq})
/* g(Node, Character) is the transition function */
1  final ← empty set
2  FOR  p = 1 TO number of patterns
3    Current ← root
4    FOR  j = 1 TO mk
5      IF  g(Current, pat[p][j]) = null
6        insert(Current , pat[p][j])
7      ENDIF
8      Current ← g(Current, pat[p][j])
9    ENDFOR
10   final ← union(final, Current)
11 ENDFOR
```

```
computation_failure (P={P1, P2, ……, Pq})
/* F(Node) is the failure function */
1  F(root) ← fail
2  FOR  ALL { Current' | parent(Curren ') = root}
3    F(Current') ← root
4  ENDFOR
5  FOR  ALL { Current' | depth(Current')>1} in BFS order
6    Current ← parent(Current')
7    c ← the symbol with g(Current , c) = Current'
8    r ← F(Current)
9    WHILE r ≠ fail AND g(r, c) = fail
10     r ← F(r)
```

```
11   ENDWHILE
12   IF r = fail
13      F(Current') ←g(root, c)
14   ELSE
15      F(Current') ← g(r, c)
16      IF isElement(h(Current'), final)
17        final ← union(final, Current')
18      ENDIF
19 ENDIF
20 ENDFOR
```

3.3.4.4 Aho-Corasick 多模式匹配算法不足

实验证明在 1 万次搜索比较中，AC 算法的查找效率明显高于 BM 算法[30]。AC 算法的时间复杂度为 O(n)，预处理的时间随模式的大小呈线性变化，其时间复杂度为 O(m)。但是，Aho-Corasick 算法在对文本进行搜索的过程中没有跳跃，而是按顺序输入文本字符，不能象 BM 算法那样跳过不必要的比较，因此在实际的搜索过程中，Aho-Corasick 算法不是性能最佳的搜索算法，并且有限自动机算法是以空间换时间的经典算法，当模式集较大时可能产生空间膨胀问题，这是因为对于一个总长度为 m 个字节的模式集，根据有限自动机的构造原理，在最坏的情况下具有 m 个状态，对于每个状态可能有 256 种不同的输入，因此描述有限自动机的二维表所需内存空间为 256*m 字节，当模式集较大时对于内存的要求将会变得很高。

3.4 算法的改进

3.4.1 改进 HAC 算法提出

BM、BMH、QS 算法是单一模式匹配，不适应越来越多的多模式同时匹配；而 AC 算法没有利用启发式策略进行跳跃，所以效率有待提高。如果能够同时利用两者优点，效率将大大提。有很多的研究者对多模 AC 算法结合单模 BM、BMH、QS 算法，使 AC 算法每次匹配跳跃的距离更大方面都作了许多的研究。改进算法的思想在于一次跳过部分不必要比较的字符数量，即在失配后模式能移动的距离。在与自动机相结合的算法中，主要考虑失配后窗口移动的距离。BM 算法是依据不匹配的文本字符来计算跳过的文本字符数；QS 算法是根据紧靠窗口最右端的下一个文本字符来计算跳过的文本字符数；而 BMH 算法则是根据对齐窗口最右端的文本字符来计算跳过的文本字符数。各算法都考虑到了跳跃，但都有其局限性，不可能每次都达到最大的跳跃距离。因此，本篇论文提出了一种

新的模式匹配算法 HAC 算法，该算法混合 BM 算法、QS 算法、BMH 算法中的跳跃距离计算方法，结合 AC 算法，使 Aho-Corasick 自动机在对文本进行搜索的过程中每次均能保持 BM 算法、QS 算法、BMH 算法三种算法跳跃的最大距离，缩短比较次数和时间。该算法预处理阶段的时间复杂度为 $O\left(\sigma+\sum_{k=1}^{q} m_k\right)$，空间复杂度为 $O(\sigma)$，匹配阶段的最坏时间复杂度为 $O(n\cdot \max len)$。

3.4.2 改进的 HAC 算法思路

改进的 HAC 算法仍使用 Aho-Corasick 自动机，将所有模式构成一个反向 Aho-Corasick 自动机，同时根据 BM 算法、BMH 算法和 QS 算法的坏字符启发，计算机 bmBc[]（BM 算法和 BMH 算法的 bmBc[]值相同），qsBc[]，当发生不匹配时，根据搜索窗口中失配的文本字符、窗口中文本的最右字符和紧靠窗口右端的下一个文本字符分别计算窗口移动量，取其中的最大值移动窗口，以消除不必要的比较。

改进的 HAC 算法如下：

（1）已知条件

设在文本串 S[0:n-1]中一次查找多个模式串 P_1，P_2，……，P_q；所有模式串形成模式串集合{P}。其中 q 为模式串的数目；模式串 P_k 的长度是 m_k，即 P_k[0:m_k-1]（$1\leqslant k\leqslant q$）；minlen 是最短模式串的长度，即 minlen=min{m_k | $1\leqslant k\leqslant q$}

（2）预处理阶段

将模式集合组成一个反向 Aho-Corasick 自动机，建立转向函数 Goto、失败函数 Failure 和输出函数 Output。

BM 算法的坏字符移动基本思想是利用本次匹配不成功，根据窗口中失配的文本字符σ，由 bmBc[σ]来决定窗口移动的距离。BMH 算法的基本思想是利用本次匹配不成功，根据窗口最右端的文本字符σ，由 bmhBc[σ]来决定窗口移动的距离。QS 算法的基本思想是利用本次匹配不成功，根据紧靠窗口最右端的下一个文本字符σ，由 qsBc[σ]来决定窗口移动的距离。分别计算 bmBc[σ]、bmhBc[σ]和 qsBc[σ]。

三个坏字符转移表 bmBc[σ]、bmhBc[σ]和 qsBc[σ]的定义域为字符集$\sigma\in\sum$，它的大小为字符集中所有字符的个数，对于需要进行中文匹配的情况，需要扩充 ASCII 字符集，数组大小则为 256，与模式集的大小无关；值域为正整数。bmBc[σ]、bmhBc[σ]和 qsBc[σ]表示字符σ在任一模式串中最右出现的位置到该模式串尾的距离，但是计算的方式略有差别，bmBc[σ]和 bmhBc[σ]的计算方法

相同，bmBc[σ]值不能超过 minlen，qsBc[σ]值不能超过 minlen+1，否则最短的模式串有可能被漏掉。

bmBc[σ]定义如下：

$$bmBc[\sigma]=\begin{cases}\min len & \sigma\text{不在}\{P\}\text{中}\\ \min\{j \mid P[m_k-1-j]=\sigma,0\le j<\min len,1\le k\le q & \sigma\text{在}\{P\}\text{中}\end{cases}$$

解释：①如果字符σ不出在模式集{P}中，bmBc[σ]取值为 minlen；②如果字符σ出在模式集{P}中，bmBc[σ]取字符σ在任一模式串中最右出现的位置到该模式串尾的距离的最小值，如果这个最小值大于 minlen，则取值为 minlen；如果字符是模式串的尾字符，则取值为 0。

qsBc[σ]定义如下：

$$qsBc[\sigma]=\begin{cases}\min len+1 & \sigma\text{不在}\{P\}\text{中}\\ \min\{j+1 \mid P[m_k-1-j]=\sigma,1\le j\le\min len,1\le k\le q & \sigma\text{在}\{P\}\text{中}\end{cases}$$

以 AC 算法的例子为例，模式集{P}={spam，stop，is}。①建立反向的 Aho-Corasick 自动机如图 3.4.2.1。

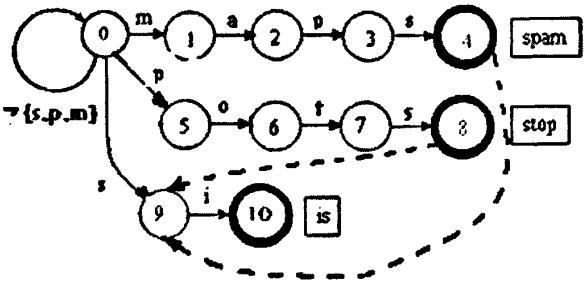

图 3.4.2.1 反向 Aho-Corasick 自动机

②计算 minlen=2。

③计算 *bmBc* 和 qs*Bc*。计算结果见表 3.4.2.1。

表 3.4.2.1　*bmBc* 和 qs*Bc* 计算值

字符 i	s	p	a	m	t	o	i	其他字符
bmBc[i]	2	2	1	2	2	1	1	2
qsBc[i]	1	1	2	1	3	2	2	3

（3）匹配阶段

文本匹配指针从 i=j=minlen-1，t=T[i]开始，从有限自动机的 0 状态出发，取出文本串的字符 t，判断 t 是否是模式尾字符，若是，则按状态转向函数 G(s, t)进入下一状态，从 T[i]字符向左继续进行状态转换，直至匹配成功或不成功。

HAC 算法在此处增加判断下一状态若为“ACSM_FAIL_STATE”值，表示不匹配，则文本指针右移距离由 bmBc-n（已匹配的字符数）、bmBc 和 qsBc 三者中最大值决定。若 t 不是模式尾字符，则文本指针右移距离由 bmBc 和 qsBc 两者中最大值决定。重新开始一次匹配. 如果成功匹配到某个模式串，则输出模式串在文本中的位置，以上述方式处理文本至末尾可以找出所有模式的出现位置。整个匹配算法可描述如下：

用 HAC 算法，在文本串“antispam antispam”中查找上面的模式集，匹配过程如下图 3.4.2.2 HAC 算法模式匹配过程图。

字符比较	i	bmBc	qsBc	BmBc-n	output
a n t i s p a m a n t i s p a m	1	BmBc[n]=2	QsBc[t]=3		
^			取较大值		
a n t i s p a m a n t i s p a m	4	BmBc[s]=2	QsBc[p]=1	0	is
^		取较大值			匹配成功
a n t i s p a m a n t i s p a m	6	BmBc[a]=1	QsBc[m]=1		
^		取较大值			
a n t i s p a m a n t i s p a m	7	BmBc[m]=2	QsBc[a]=2	-3	spam
^		取较大值			匹配成功
a n t i s p a m a n t i s p a m	9	BmBc[n]=2	QsBc[t]=3		
^			取较大值		
a n t i s p a m a n t i s p a m	12	BmBc[s]=2	QsBc[p]=1	0	is
^		取较大值			匹配成功
a n t i s p a m a n t i s p a m	14	BmBc[a]=1	QsBc[m]=1		
^		取较大值			
a n t i s p a m a n t i s p a m	14	BmBc[a]=1	QsBc[m]=1		
^		取较大值			
a n t i s p a m a n t i s p a m	15			-3	spam
^					匹配成功

图 3.4.2.2　HAC 算法模式匹配过程图

根据以上改进的 HAC 匹配算法，具有这样的特点：

①多模式匹配

应用了 AC 算法的匹配自动机原理，将所有模式构成自动机状态转换表，扫描一次文本实现多个模式的匹配。同时又充分利用了入侵检测规则中许多规则

拥有相同字符前缀的这个特点，大大减少了单模式算法中重复的没有必要的比较。

②跳跃移动

采用了单模式匹配算法中的跳跃匹配思想，在每次失配后根据相应规则产生的跳跃移动文本匹配指针，再进行下一次的匹配。解决了 AC 算法中文本匹配指针没有跳跃，而是一个字符一个字符的输入进行比较的问题，从而也减少不必要的字符比较。

③偏移量大

引入了单模式的 BM 算法、BMH 算法和 QS 算法的跳跃思想，同时考虑三个字符的影响，而且取三者跳跃的最大值，跳跃距离大，即最大跳跃可以达到 minlen+1 个字符。

④简化跳跃规则

首字检测字符是否是模式串尾字符，是尾字符，则调用自动机状态转换函数；在非尾字符位置失配情况下，采用了类似 BM 算法、BMH 算法和 QS 算法的坏字符移动启发，同时考虑三个字符但不是作为字符串去处理，大大节省了算法的运行时间。

经过这样的改进后，HAC 算法在匹配过程失配处很快得到跳跃值，并且得到了较大的偏移量，在每次失配后，模式集移动的距离加大，就可以减少总的字符比较次数，从而达到了快速匹配的目的。

3.4.3 改进的 HAC 算法实现

H-Aho- Corasick($P=\{P_1, P_2, \cdots\cdots, Pq\}$, $T= t_1t_2\cdots\cdots t_n$)

```
1  Preprocessing
2  AC ← Build_trie (P^r={P_1^r, P_2^r, ……, Pq^r}) and bmBc()、qsBc()
3  AC ← computation_failure(P)
4  Build_ bmBc_ qsBc (P={P_1, P_2, ……, Pq})
5  Searching
6  Current ← root
7  while i = minlen and i< n do
8     j ← i
9    IF  text[j]∈ end character of patterns
10      while Current ← g(Current, text[j--])≠ fail
11         IF isElement(Current, final)
12             RETURN TRUE
```

```
13          ENDIF
14        END while
15    else
16        i=i+max(bmBc[j--]-i+j, bmBc[text[i]], qsBc[text[i+1]])
17    ENDIF
18 END while
```

---

build_trie($P^v=\{P_1^{rv}, P_2^{rv}, \cdots\cdots, Pq^{rv}\}$)

```
/* g(Node, Character) is the transition function */
1  final ← empty set
2  FOR  p = 1 TO number of patterns
3     Current ← root
4     FOR  j = 1 TO mk
5        IF  g(Current, pat[p][j]) = null
6           insert(Current , pat[p][j])
7        ENDIF
8        Current ← g(Current, pat[p][j])
9     ENDFOR
10    final ← union(final, Current)
11 ENDFOR
```

---

computation_failure($P^v=\{P_1^{rv}, P_2^{rv}, \cdots\cdots, Pq^{rv}\}$)

```
/* F(Node) is the failure function */
1  F(root) ← fail
2  FOR  ALL { Current' | parent(Curren ') = root}
3     F(Current') ← root
4  ENDFOR
5  FOR  ALL { Current' | depth(Current')>1} in BFS order
6     Current ← parent(Current')
7     c ← the symbol with g(Current , c) = Current'
8     r ← F(Current)
9     WHILE r ≠ fail AND g(r, c) = fail
10       r ← F(r)
11    ENDWHILE
```

```
12   IF r = fail
13      F(Current') ←g(root, c)
14   ELSE
15      F(Current') ← g(r, c)
16      IF isElement(h(Current'), final)
17        final ← union(final, Current')
18      ENDIF
19 ENDIF
20 ENDFOR
```

```
Build_ bmBc_ qsBc (P={P1, P2, ……, Pq})
1  FOR c∈Σ do
2    bmBc[c] ← minlen
3    qsBc[c] ← minlen+1
4  ENDFOR
5  FOR  p ∈ 1 … number of patterns
6     FOR  j∈ 0 … mk-2 do
7       IF  mk - j-1< bmBc [pkj]≤minlen
8         bmBc [pkj] ← mk - j-1
9       ENDIF
10    ENDFOR
11    FOR  j∈ 0 … mk-1 do
12      IF  mk - j< qsBc[pkj]≤minlen
13        qsBc[pkj] ← mk - j
14     ENDIF
15    ENDFOR
```

3.5 HAC 算法复杂性综合分析

设字母表的大小为Σ，所有模式串的长度之和$\sum_{k=1}^{q} m_k$。

3.5.1时间复杂度(time complexity)分析

算法的时间复杂度又称计算复杂度(computation complexity)，是一个算法运行时间的相对量度值。

1. 预处理阶段的时间复杂度

预处理阶段goto函数的构造的时间与$\sum_{k=1}^{q} m_k$成线性关系，时间复杂度为$O\left(\sum_{k=1}^{q} m_k\right)$。

bmBc函数分两步:①初始化的计算量与Σ成线性关系。②更新bmBc函数值要扫描每个模式串，其计算量与$\sum_{k=1}^{q} m_k$成线性关系，所以bmBc函数的时间复杂度为$O\left(\sigma+\sum_{k=1}^{q} m_k\right)$。

qsBc函数的计算与bmBc函数类似,其时间复杂度为$O\left(\sigma+\sum_{k=1}^{q} m_k\right)$。

综上所述，在预处理阶段HAC算法的时间复杂度为$O\left(\sigma+\sum_{k=1}^{q} m_k\right)$。

2. 查找阶段的时间复杂度

模式串集合中，模式串的最小长度为minlen，最大长度为maxlen。

在最优情况下算法时间复杂度为$O\left(\frac{n}{\min len+1}\right)$，在最坏情况下，算法时间复杂度为$O(n\cdot \max len)$。

平均情况下，时间复杂度的分析比较复杂，而且与字符的出现概率有关，因此需要通过概率模型进行计算。通常算法最好情况的复杂度没有什么实际意义，而通过最坏情况下的复杂度可以估算到算法运行时所需要的相对最长时间。最有实际意义的是平均情况下的复杂度，它反映了运行一个算法的平均快慢程度。

3.5.2空间复杂度(space complexity)分析

空间复杂度是对一个算法在运行过程中占用的存储空间大小的量度，包括存储算法本身所用的存储空间，算法的输入输出数据所用的存储空间和算法在运行过程中临时占用的存储空间等。

多模式匹配算法为了加快匹配阶段的速度，都要在执行匹配之前对模式集进行预处理，构造某种数据结构，为匹配过程提供支持。HAC算法基于AC算法，需要对模式状态进行存储，另外还需要两个跳跃数组bmBc[ch]和qsBc[s]。算法需要的空间为(256*m+256+256))个内存单元，(其中m是自动机的状态总数)。

总的来说，HAC算法消耗较多的内存空间,随着模式数量的增大，模式状态将

会增大，所需内存空间也会急剧膨胀。但在实际的网络入侵检测系统中，检测攻击的实时性是主要的要求，因此匹配时间始终是考虑的首要因素，以空间的消耗换取时间效率的提高是可行的。同时，随着存储技术的发展，计算机内存容量加大，内存的价格也呈现越来越低的趋势，内存开销也会越来越变成次要因素。

3.6 本章小结

本章首先介绍了网络入侵检测系统中重要的单模式 BM 算法、BMH 算法、QS 算法及经典的多模式 AC 算法，在此基础上详细地介绍了一种改进的多模式匹配算法 HAC 算法，并对该算法进行了性能分析。该算法是在 AC 算法的基础上，结合 BM 算法、BMH 算法和 QS 算法思想产生跳跃，通过对搜索窗口右端内外和失配处的文本三个字符来决定窗口移动的距离，简化了匹配过程，能在实时的入侵检测中发挥很好的作用。

4 模式匹配算法测试

从科学研究的角度来看，任何一种理论都需要通过实践的检验。入侵检测系统评测的基本目标，一方面能够证实 IDS 的有效性；另一个方面，评测也能反映 IDS 的缺陷和不足[45]，明确改进的方向，所以 IDS 评测对于研究人员而言非常重要。测试主要是基于对算法的有效性测试和与其它算法的性能比较方面的综合测试。有效性测试主要的测试目标是检验入侵检测系统是否达到了设计目标，比如对于基于特征的入侵检测系统，它应该能够准确无误地检测到其特征集所对应的所有攻击在进行。算法性能对比的综合测试的目标是希望检验文本所提出的新的匹配算法在入侵检测系统应用中的优势。目前对 IDS 的测试还没有统一的标准。

4.1 测试环境

本章进行模式匹配算法的测试，测试环境为：

(1) 操作系统WindowsXP。

(2) 系统配置为2.4GHz，Intel CPU，256M RAM。

(3) 算法用C语言实现，编译器采用VC++6.0。

4.2 测试过程

测试的目的主要是针对本文提出的算法在匹配时间方面与 AC 算法、ACBMH 算法的性能比较。选取 6.7M 的字符文本作为数据集，在其中查找多个模式串，找出所有模式出现的位置。从文本中选取不同长度的模式串，测试在不同模式集和不同模式长度下算法的性能。为了方便比较，实验中每一组模式串长度一致，选取的模式能够匹配和不能匹配比例为 7：3。

实验共测试两个方面的内容：

1. 几种算法匹配时间随模式串长度增加时的变化情况。模式集中模式数目固定为 5 个，模式长度从 10 到 100 以 10 为单位递增。

2. 几种算法匹配时间随模式集大小增加时的变化情况。模式集中模式长度固定为 40，模式的数目从 1 到 40 以 8 为单位递增。

测试结果见表 4.2.1 和表 4.2.2。

表 4.2.1　算法匹配时间随模式串长度变化情况　　（单位：S）

算法 / 模式长度	AC	AC_BMH	HAC
10	0.63	0.51	0.54
20	0.62	0.52	0.52
30	0.63	0.53	0.56
40	0.62	0.54	0.53
50	0.63	0.51	0.54
60	0.62	0.52	0.53
70	0.61	0.52	0.53
80	0.62	0.53	0.52
90	0.63	0.52	0.51
100	0.64	0.52	0.51

表 4.2.2　算法匹配时间随模式串数目变化情况　　（单位：S）

算法 / 模式串数目	AC	AC_BMH	HAC
1	0.61	0.52	0.51
8	0.62	0.57	0.56
16	0.63	0.58	0.58
24	0.63	0.59	0.59
32	0.64	0.60	0.59
40	0.65	0.59	0.59

4.3 测试结果分析

将测试结果表分别以图形表示，如图 4.3.1 算法匹配时间随模式串长度变化情况和图 4.3.2 算法匹配时间随模式串数目变化情况。

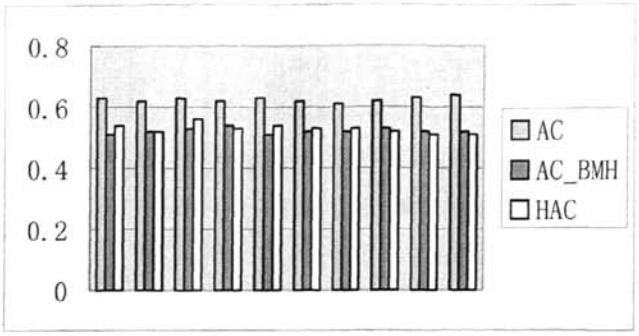

图 4.3.1 算法匹配时间随模式串长度变化情况。

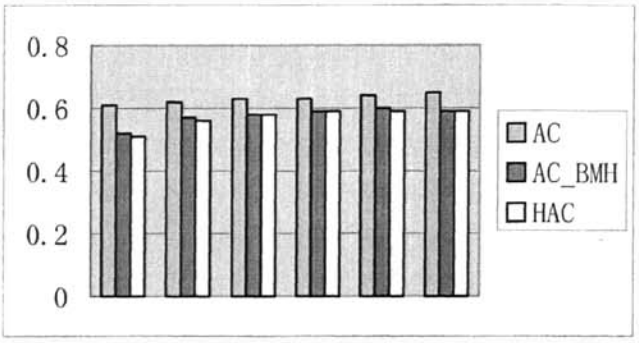

图 4.3.2 算法匹配时间随模式串数目变化情况

分析测试结果，可以得出如下结论：

（1） 当数据量和模式数目一定时，随着模式长度的增加，3 种算法的匹配时间变化不明显。因为几种匹配算法只对文本扫描一次，由于模式长度有限，指针回溯的情况与文本串长度相比可忽略，因此随模式长度的增加，匹配时间增加不明显。由此可见，当数据量和模式数目一定时，各个算法的匹配时间，随着模式长度的增加变化不明显。

（2） 当数据量和模式长度一定时，随着数量的增加，3 种算法的匹配时间变化不明显。因为几种匹配算法只对文本扫描一次，由于模式数量有限，指针回溯的情况与文本串数量相比可忽略，因此随模式数量的增加，匹配时间增加不明显。由此可见，当数据量和模式长度一定时，各个算法的匹配时间，随着模式数量的增加变化不明显。

（3） AC_BMH 算法和 HAC 算法在模式串长度变化和模式串数目变化时，都比 AC 算法有更好的性能。而 AC_BMH 算法和 HAC 算法在模式串长度变化和模式串数目变化时，性能相差不明显。这也符合三种算法的设计思想，

AC 算法在模式失配时，窗口沿文本挨个字符移动，不能跳过不必要的匹配；而 AC_BMH 算法在模式失配时只根据窗口最右端文本字符来决定窗口跳跃距离；HAC 算法在模式失配时根据三种情况中的最大值来决定窗口跳跃距离，即失配的文本字符、窗口最右端文本字符和窗口最右端下一个文本字符三种情况，当选取的模式集所含字符数较少时，HAC 算法与 AC_BMH 算法相比优势并不明显。

（4） 实验结果表明，HAC 算法需要的匹配时间是 AC 算法的 90%左右。

综上所述，改进的 HAC 算法在模式数量和最短模式长度变化时，其检测速度优于 AC 算法，在性能面优于 AC 算法，可改善入侵检测系统的性能。

5 结束语

网络是一把双刃剑，它给人类生活带来无限美好的同时，也因其潜在的不安全因素给人类带来了无限烦扰。特别是随着网络应用日益深入人类的日常生活，网络安全问题就越来越突出，特别在金融、电信、商务等系统的应用中。所谓道高一尺，魔高一丈，各种入侵检测技术也在飞速发展，而且越来越受到人们的重视。

5.1 本文的主要工作

通过对网络入侵检测技术、入侵检测模型、还有入侵检测系统 Snort 的研究和分析，对网络入侵检测的实现方法，实现架构都有了很好的理解。本文重点分析研究了开放源码入侵检测系统 Snort 的检测引擎采用的模式匹配算法，介绍了单模匹配 BM 算法、BMH 算法、QS 算法和多模匹配的 AC 算法，在分析各种算法的基础上，本篇论文提出了一种新的模式匹配算法 HAC 算法，该算法混合 BM 算法、QS 算法、BMH 算法中的跳跃距离计算方法，结合 AC 算法，使 Aho-Corasick 自动机在对文本进行搜索的过程中每次均能保持 BM 算法、QS 算法、BMH 算法三种算法跳过不必要比较的的字符数最大，缩短比较次数和时间，提高了入侵检测系统的检测效率。

5.2 进一步改进及展望

随着网络应用的扩展，网络入侵检测已经成为当今的一个研究热点。网络入侵检测效率的提高，归根结底就是模式匹配算法效率的提高，因此多模式匹配算法的研究仍然是一个备受关注的热点问题。另外入侵检测系统受模式串的影响很大，所以如何选择合适有效的特征串，也是提高算法查找速度需要考虑的问题。

通过对入侵检测理论的研究和开放源码的入侵检测系统 Snort 的深入分析研究，本人从中学习到了许多知识，但由于本人水平和时间有限，所提出改进的 HAC 多模式匹配算法还不是最快的匹配算法，而且随着模式数量的增多，其所要求的内存空间会发生膨胀现象，因此研究时间、空间相对较优的算法，既可以最大限度的提高系统效率，又可尽量避免发生内存膨胀现象，从而实现在大流量情况下的实时检测势在必行。

由于研究水平和研究时间的限制，本课题在对 Snort 规则匹配的改进上还较薄弱，还有许多有待完善和值得继续探讨的问题。

文中提出的改进的 HAC 多模式匹配算法不能够从理论角度来证明它的优越性，而且也没有在实时的入侵检测系统中进行测试，并且其检测速度并没有大幅度的提高。在下一步的工作中，可以继续对其他一些优秀的多模式匹配算法进行分析研究，从中寻找更多优秀的算法。如 SBOM 算法，该算法的最大优点建

立的 Backward Oracle 自动机在对文本进行的扫描过程中，无需回溯，检测速度也较其他算法在模式集数量较多及模式长度较长的情况下具有优势。

由于本人水平有限，文中难免有不足和错误之处，恳请老师和同学批评指正。

致 谢

对于我这样年龄的学生，再次进入学校进行系统的学习研究是一种奢侈，在繁忙的工作、学习、家庭和生活之间轮转，既感到时间紧张，也感到生活充实。随着学位论文的完成，这样的紧张生活也即将告一段落，回顾这段岁月，心中不禁感慨万千，在过去的这段岁月里，我要感谢所有帮助过我的老师和朋友。

首先我要感谢的是我的导师兰少华教授，兰老师学识渊博、思维开阔，谆谆教导，对我的毕业论文倾注了不少的心血，他严谨的学术风格深深的影响着我使我收益无穷。在论文完成之际，向我的导师致以最诚挚的敬意。

其次我要感谢南京理工大学所有教授过我课程的老师，他们认真的教学态度和教学方法也深深地影响着我。

感谢所有在 Snort 系统开发研究方面的专家和在模式匹配方面的研究专家，是他们提出的各种思想开阔了我的眼界，给了我极大的启发。

感谢我工作单位的领导和同事，为我的学习提供了方便和支持。

特别感谢一位默默鼓励与支持我的朋友。

感谢南京航空航天大学的李渝萍老师，特别在收集资料、查阅借阅资料方面，尽她最大的努力帮助我、鼓励我。

感谢开源入侵检测系统 Snort 的作者和大量的不知名的网络作品提供的宝贵资源。

最后要感谢我的家人，是他们的鼓励和支持才能使我毫无后顾之忧地投身到学习工作中，他们对我的无私的支持、理解和信任伴随着我走过了人生的岁月，他们是我的精神支柱，他们是我的最大骄傲，没有他们，我无法完成今天的学业，也就没有今天的我。

最后，祝愿南京理工大学明天更加辉煌。

参考文献

1 金山毒霸. 中国 2007 年上半年病毒疫情及互联网安全报告[R]. http://www.kingsoft.com/news/safenews/2007/07/04/110803.shtml

2 2007年网络安全技术发展分析[R].http://www.hacker.cn/News/aqxw/2007-5-4/075414404754H3970JDIB1421JA.shtml

3 Jack Koziol.Snort 入侵检测实用解决方案[M] .北京：机械工业出版社，2005

4 [美]Brian Caswell, JayBeale, James C.Foster 著.宋劲松等译.Snort2.0 入侵检测[M].北京：国防工业出版社，2004

5 宋江静.入侵检测技术之探讨[DB/OL]. http://www.sdca.gov.cn/hangye/elec_book/2005-12/web/12-1.htm

6 [美]Rebecca Gurley Bace 著.陈朋奇，吴秋新，张振涛等译.入侵检测[M].北京：人民邮电出版社，2001

7 宋劲松编著.网络入侵检测——分析、发现和报告攻击[M].北京：国防工业出版社，2004

8 吴婷. 网络入侵检测系统的研究现状与发展趋势[J]. 中共郑州市委党校学报，2006(4):129-132

9 周军民.IDS 技术研究历史[DB/OL]. http://www.bitscn.com/cisco/ids/200604/13893.html

10 韩东海，王超，李群编著. 入侵检测系统实力剖析[M]. 北京：清华大学出版社，2002

11 余兆力. 基于 Snort 的网络入侵检测系统研究[D]. 浙江：浙江工业大学，2004

12 谭汉松.一种新的快速多模式匹配算法[J].计算机工程，2005(18):119-121

13 R.S.Boyer and J.S.Moore.A Fast String Searching Algorithm[J]. Communications of ACM 20(10),1977:762-772

14 R.N.Horspool.Practical fast searching in strings[J].Software : Practice and Experience,10(6) 1980:510-516

15 D.M.Sunday. A very fast substring search algorithm[J].Communications of the ACM,33(8) 1990:132-142

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

17 Jason Coit, Stuart Staniford, Joseph McAlerney. Towards faster patern matching for intrusion detection or exceeding the speed of

snort [C]. In:DARPA Information Survivability Conference and Exposition,2001

18 殷丽华，方滨兴.一种改进的多模式匹配算法[J] .华中科技大学学报(自然科学版)，2005(12):300-303

19 蔡晓妍，戴冠中，杨黎斌.改进的多模式字符串匹配算法[J].计算机应用，2007(6):1415-1417

20 王永成，沈州，许一震. 改进的多模式匹配算法[J].计算机研究与发展，2002(1):55-60

21 许一震，王永成，沈州.一种快速的多模式字符串匹配算法[J].上海交通大学学报,2002(4):516-520

22 黎鹰,李亮.入侵检测系统的现状与发展[DB/OL].http://www.gotoread.com/article/?NewID=86E31325-3B8A-4E4E-84BD-C788A9EA0F3C

23 黄金莲. 网络入侵检测系统中模式匹配算法的研究[D].河北：华北电力大学，2005

24 Snort 相关技术网站（源代码、规则等）. http://www.snort.org

25 宋俊承. 基于网络的入侵检测系统中字符串匹配算法的应用研究[D].辽宁：东北大学，2005

26 唐正军. 入侵检测技术导论[M].北京：机械上业出版社，2004

27 空竹林. snort 源码分析系列之一附件[DB/OL].http://blog.chinaunix.net/u/29331/showart_368693.html

28 国家计算机网络应急技术处理协调中心. 2007 年上半年网络安全工作报告[R]. http://www.cert.org.cn/UserFiles/File/CNCERTCC200701.pdf

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

30 胡德华.Snort 检测引擎的改进与实现[D].辽宁：东北大学，2005

31 宋明秋，张国权，邓贵仕. IDS 中新的快速多模式匹配算法及其设计[J]. 计算机工程与应用 2005(21):159-162

32 康振勇. 网络入侵检测系统 Snort 的研究与改进[D].陕西:西安电子科技大学 2006

33 [美]Gonzalo Navarro,Mathieu Raffinot.柔性字符串匹配[M].北京：电子工业出版社，2007

34 Christian Charras-Thierry Lecroq.EXACT STRING MATCHING ALGORITHMS [DB/OL].http://www-igm.univ-mlv.fr/~lecroq/string/

35 http://www.ll.mit.edu/IST/ideval/data/1999/1999_data_index.html

[DB/OL]

36 赵念强，鞠时光. 入侵检测系统中模式匹配算法的研究[J].微计算机信息(管控一体化),2005(14):23-26

37 [美]Jack Koziol 著.吴溥峰，孙默，许诚等译.张玉清审.Snort 入侵检测实用解决方案[M] .北京：机械工业出版社，2005

38 Pekka Kilpelainen. Lecture 4:Set Matching and Aho-Corasick Algorithm. Molecular Sequence Algorithms,Spring 2004. http://www.cs.uku.fi/~kilpelai/BSA04/lectures/print04.pdf

39 Boyer-Moore Algorithm [DB/OL] . http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/boyerMoore.htm

40 Boyer-Moore-Horspool-Sunday 算法介绍 DB/OL] . http://www.cublog.cn/u/22679/showart_223817.html

41 字符串查找算法--Sunday 算法 [DB/OL] . http://saraven.nklog.org/post/2006/12/01/ucdumcui

42 AC 多模匹配算法小结 [DB/OL] . http://blog.csdn.net/lindan1984/archive/2006/12/20/1450307.aspx

43 周军民.IDS 技术研究历史[DB/OL]. http://www.bitscn.com/cisco/ids/200604/13893.html

44 中国网管联盟[DB/OL].http://www.bitscn.com/cisco/ids/index.html

45 史美林，钱俊，许超. 入侵检测系统数据集评测研究[J]. 计算机科学，2006(8):43-45

46 KDD Cup 1999 Data .http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html[DB/OL].