

2010 年全国大学生信息安全竞赛

作品报告

作品名称: AegisShield2.0-基于多核处理平台的网络行为监控系统

参赛学校: 四川大学

学院/系: 软件学院

指导教师: 梁刚

组 长: 萧国翘

组 员: 萧国翘、张伟、姚锦华、钟存伟

通信地址: 四川大学软件学院 07 级 169 信箱

电 话: 18608007022

电子邮箱: isfan@live.cn

提交日期: 2010-6-30

填写说明

1. 所有参赛项目必须为一个基本完整的设计。作品报告书旨在能够清晰准确地阐述（或图示）该参赛队的参赛项目（或方案）。
2. 作品报告采用 A4 纸撰写。除标题外，所有内容必需为宋体、小四号字、1.5 倍行距。
3. 作品报告中各项目说明文字部分仅供参考，作品报告书撰写完毕后，请删除所有说明文字。(本页不删除)
4. 作品报告模板里已经列的内容仅供参考，作者也可以多加内容。

目 录

第一章 摘要.....	1
1.1 项目摘要.....	1
1.2 定义与缩写.....	2
1.3 运行环境.....	4
第二章 作品介绍.....	5
2.1 项目背景.....	5
2.2 作品特点.....	8
2.3 主要功能概述.....	10
2.3.1 入侵检测功能.....	11
2.3.2 应用层协议分析.....	12
2.3.3 输出缓存.....	14
2.3.4 自定义规则.....	15
2.3.5 数据库.....	16
2.3.6 WEB 登陆.....	16
2.3.7 查看日志.....	17
2.3.8 管理规则.....	17
2.3.9 管理用户.....	18
2.3.10 系统信息.....	18
2.4 概要实现方案.....	19
2.5 应用市场分析与后期开发.....	21
第三章 实现方案.....	22
3.1 总体设计.....	22
3.1.1 系统构架.....	22
3.1.2 技术路线.....	23
3.1.3 模块结构.....	24
3.1.4 处理流程.....	25
3.1.5 流水线规划.....	27
3.1.6 数据库设计.....	28

3.2 模块概要设计	32
3.2.1 零拷贝模块	32
3.2.2 应用层协议分析模块	35
3.2.3 多核优化模块	39
3.2.4 WEB 模块	40
3.3 模块详细设计	44
3.3.1 零拷贝模块	44
3.3.2 应用层协议分析模块	70
3.3.3 多核调度模块	93
3.3.4 WEB 模块	99
第四章 性能测试	110
4.1 测试方案	110
4.2 测试环境	110
4.3 环境搭建	111
4.4 测试用例及结果	117
4.4.1 应用层协议分析模块	117
4.4.2 零拷贝部分	131
4.5 多核调度分析	140
4.6 WEB 功能测试	144
第五章 创新性	155
第六章 总结	156
参考文献	157

第一章 摘要

1.1 项目摘要

传统的入侵检测系统存在的缺陷：漏报误报率高，丢包率高，自身不够安全等情况，**AegisShield2** 基于多核平台的网络行为分析系统期望通过一系列措施改善这些缺陷，以提高入侵检测系统的可信度和健壮性，在提供一套可靠的入侵检测功能同时还额外增加了内部用户网络行为的分析，包括用户使用的网络软件分析、浏览网站分析等。可以说，**AegisShield2** 把外部的行为和内部的行为分析有机地融合在一起，为管理员提供网络安全分析的有力工具。相对其他的同类系统，本系统有以下特点：

1)使用零拷贝技术

减少数据拷贝和共享总线操作的次数，消除通信数据在存储器之间不必要的中间拷贝过程，降低系统调用的开销，有效地提高抓取数据包的效率，支持千兆网络；

2) 使用多核编程技术

充分发挥现有多核处理器的能力，系统同一时刻能够并行处理多个数据包，较大提高了入侵检测系统的工作效率，弥补了现有入侵检测系统因串行处理不及而丢失数据包的问题；

3) 应用层行为分析

通过使用应用层传输数据特征库，快速识别出多达十余中国内最常见的网络软件，如 QQ、P2P、HTTP 等；

4) 基于 B/S 的友好界面

通过友好简易的操作界面，网络管理员只需要简单地培训，就可以有效地使用。更重要的是，管理员可以在任意有浏览器支持的平台上登录 **WEB** 服务器操作。我们使用了分级 **WEB** 管理，以适应不同的使用角色；

1.2 定义与缩写

AegisShield

简称 **AS**，本软件系统的名称，为一个经过多核优化，运行于 **Linux** 下，对网络行为（网络攻击、网络软件使用）等进行监控的系统，具有安全性、可靠性、高效性。具体内容请参看 2.3 章节。

IDS

IDS 是英文 “Intrusion Detection Systems” 的缩写，中文意思是 “入侵检测系统”。专业上讲就是依照一定的安全策略，对网络、系统的运行状况进行监视，尽可能发现各种攻击企图、攻击行为或者攻击结果，以保证网络系统资源的机密性、完整性和可用性。

Zero-Copy

简称 **ZC**，零拷贝（zero-copy）是实现主机或路由器等设备高速网络接口的主要技术。零拷贝技术通过减少或消除关键通信路径影响速率的操作，降低数据传输的操作系统开销和协议处理开销，从而有效提高通信性能，实现高速数据传输。

Snort

开放源代码的入侵检测系统 **Snort**. 直至今天, **Snort** 已发展成为一个多平台 (Multi-Platform), 实时 (Real-Time) 流量分析, 网络 IP 数据包 (Pocket) 记录等特性的强大的网络入侵检测/防御系统 (Network Intrusion Detection/Prevention System), 即 **NIDS/NIPS**. **Snort** 符合通用公共许可 (GPL), 本系统的核心部分基于 **Snort**。

Rule

规则，为特征检测提供检测依据。特征检测 (Signature-based detection) 又称 **Misuse detection**，这一检测假设入侵者活动可以用一种模式来表示，系统的目标是检测主体活动是否符合这些模式。它可以将已有的入侵方法检查出来，但对新的入侵方法无能为力。其难点在于如何设计模式既能够表达 “入侵” 现象又不会将正常的活

动包含进来。本软件系统使用这检测方法

Packet

封包、数据包。数据要在通讯系统中必须要先经过某些处理，才能在网络当中传递，例如将数据切割为数个区块之后，才能在网上依照某种通讯协议来传送，这种过程就好像将包裹打包一样，称为分封，因为网络数据包又称为封包。

Detect Server

本系统的部署组成部分之一，主要完成 **Packet** 抓取与分析过程，其处于一个较安全的网络当中，其中一个网络接口被设成只进行 **Packet** 抓取，另一网络接口与 **WebUI Server** 相连，传输数据。

WebUI Server

本系统的部署组成部分之一，为用户提供 **Web** 接口，方便查看。同时与 **Detect Server** 和所在 **Intranet** 相连。为了确保 **Detect** 的安全性，避免 **WebUI Server** 受到攻击后 **Detect Server** 被入侵，它们中间由一个防火墙相连。

1.3 运行环境

要求环境	内容
操作系统	Linux 2.6.18 以上
内存	1GB RAM
支持网卡	Pcnet32 igb, for Intel 82575 ixgbe, for Intel 82598 bnx2, for BCM Server GE sky2,Marvell series. 注：要求有两块不同型号的网卡，至少有一块为上述型号。
WEB 要求	Apache2, php5.2 以上，GD2.0 以上

说明：

由于零拷贝模块在初始化时须占用大量的内核内存，以便数据包不被丢弃，在保证系统的性能与稳定性，推荐使用更高的内存。

零拷贝是依赖网络接口（网卡）驱动的，因此仅此支持部分驱动。

考虑到兼容性，Web 端推荐使用 php5.2 + GD2.0(须为内置版本)，GD 用于图形化输出检测结果。

第二章 作品介绍

2.1 项目背景

在互联网告诉发展的今天，计算机已经成为人们生活和工作的一部分，人们习惯用网络来查询，交流，购物和办公。当越来越多的公司将其核心业务向互联网转移的时候，网络安全作为一个无法回避的问题摆在人们面前。公司一般采用防火墙作为安全的第一道防线。而随着攻击者技能的日趋成熟，攻击工具与手法的日趋复杂多样，单纯的防火墙策略已经无法满足对安全高度敏感的部门的需要，网络的防卫必须采用一种纵深的、多样的手段。与此同时，目前的网络环境也变得越来越复杂，各式各样的复杂的设备，需要不断升级、补漏的系统使得网络管理员的工作不断加重，不经意的疏忽便有可能造成重大的安全隐患。在这种情况下，入侵检测系统 IDS（Intrusion Detection System）就成了构建网络安全体系中不可或缺的组成部分。

入侵检测系统是对防火墙有益的补充，入侵检测系统被认为是防火墙之后的第二道安全闸门，对网络进行检测，提供对内部攻击、外部攻击和误操作的实时监控，提供动态保护大大提高了网络的安全性，但是入侵检测系统仍缺乏对正常应用层协议的分析与识别。

在当前网络环境中，入侵检测系统存在以下几个不容忽视的问题：

随着网络技术的发展，攻击者的攻击手段也在不断地增多，手法也日趋地复杂。现有的入侵检测系统一般多是使用特征匹配的方法。然而，特征匹配不能够对未知的攻击进行防范，这也就导致了入侵检测系统的误报率和漏报率普遍偏高。虽然有许多诸如神经网络、专家系统等允许入侵检测系统进行自学习的理论研究正在进行，但效果并不理想，所以市面上成熟的入侵检测系统产品，诸如 Snort 和 bro,都以特征匹配居多。

由于入侵检测系统需要保护整个网段，在设置为混杂模式后，网段内所有的数据包会经过它。这时，如果没有良好的硬件和软件支持，IDS 常常会因为无法对海量的数据进行处理导致丢包的现象产生。这些被丢弃的数据包中，很有可能包含着会对系统安全构成威胁的包，这样也就导致了入侵检测系统的准确性下降。

入侵检测系统也是一个网络中的系统，所以它也存在着安全上的漏洞。许多网络攻击者，正是抓住了这些漏洞，首先对入侵检测系统进行攻击，使其瘫痪不能工作。这样，整个网络就进入了无保护的状态。入侵检测系统也失去了它的作用。所以，入侵检测系统自身的安全性也是信息安全领域的热点问题。

入侵检测作为一项重要的计算机网络安全技术日益受到人们的重视。目前国内外众多专家学者致力于入侵检测技术的研究，并且已经取得了不少的成果。然而目前存在的入侵检测系统在响应能力和交互能力上存在一定的不足，其中一些关键算法和技术还不够成熟。

但是，传统的入侵检测系统已经不适应时代的发展，像 **snort** 在多核环境下，不能获得很好的提高。而从应用需求上去看，越来越多的用户在使用过程中都会涉及到多任务应用环境，日常应用中用到的非常典型的有两种应用模式。

一种应用模式是一个程序采用了线程级并行编程，那么这个程序在运行时可以把并行的线程同时交付给两个核心分别处理，因而程序运行速度得到极大提高。这类程序有的是为多路工作站或服务器设计的专业程序，例如专业图像处理程序、非线性视频编辑程序、动画制作程序或科学计算程序等。对于这类程序，两个物理核心和两颗处理器基本上是等价的，所以，这些程序往往可以不作任何改动就直接运行在双核电脑上。

还有一些更常见的日常应用程序，例如 **Office**、**IE** 等，同样也是采用线程级并行编程，可以在运行时同时调用多个线程协同工作，所以在双核处理器上的运行速度也会得到较大提升。例如，打开 **IE** 浏览器上网。看似简单的一个操作，实际上浏览器进程会调用代码解析、**Flash** 播放、多媒体播放、**Java**、脚本解析等一系列线程，这些线程可以并行地被双核处理器处理，因而运行速度大大加快（实际上 **IE** 浏览器的运行还涉及到许多进程级的交互通信，这里不再详述）。由此可见，对于已经采用并行编程的软件，不管是专业软件，还是日常应用软件，在多核处理器上的运行速度都会大大提高。

日常应用中的另一种模式是同时运行多个程序。许多程序没有采用并行编程，例如一些文件压缩软件、部分游戏软件等等。对于这些单线程的程序，单独运行在多核处理器上与单独运行在同样参数的单核处理器上没有明显的差别。但是，由于日常使用的最最基本的程序——操作系统——是支持并行处理的，所以，当在多核处理器上

同时运行多个单线程程序的时候，操作系统会把多个程序的指令分别发送给多个核心，从而使得同时完成多个程序的速度大大加快。

另外，虽然单一的单线程程序无法体现出多核处理器的优势，但是多核处理器依然为程序设计者提供了一个很好的平台，使得他们可以通过对原有的单线程程序进行并行设计优化，以实现更好的程序运行效果。

AegisShield 2 基于多核处理平台的网络行为监控系统就是在这样的背景下产生的，通过使不同的检测系统协同工作不仅仅进行入侵检测，还能对应用层的协议进行分析，识别出各种网络软件，通过使用多核 **CPU** 来降低丢包率和提升处理速度。极大地改善了上面所说的几个问题。

2.2 作品特点

AegisShield 把外部的行为和内部的行为分析有机地融合在一起，为管理员提供网络安全分析的有力工具。相对其他的同类系统，它不仅仅只是一个入侵检测系统，因此，我们被它命名为网络行为监控系统，本系统有以下特点：

1、适应高速网络的零拷贝技术

现在，大部分 IDS 传统开发的入侵检测系统大多使用 **libpcap** 进行数据包的截取，在高速的网络下面，很容易造成丢包等问题。本系统通过使用零拷贝技术减少数据拷贝和共享总线操作的次数，消除通信数据在存储器之间不必要的中间拷贝过程，降低系统调用的开销，有效地提高抓取数据包的效率，支持千兆网络；

2、多核并行技术

在高速网络环境下，尽管使用零拷贝技术可以解决抓取数据包的效率问题，但是系统的瓶颈还是位于数据包的分析。在使用多核心处理器下带来了一个新的问题，在处理精度提升的状况下，对数据包的处理时间也大大加长，如果仍然使用传统的串行处理模式，必然导致大量的数据包被丢弃，进一步导致丢包率提高，这是不能容忍的。为了解决这一情况，**AegisShield** 将采用多核并行处理的方式，分别将不同的检测引擎的线程绑定在不同的 CPU 核心上，成倍提高处理效率，充分发挥现有多核处理器的能力，系统同一时刻能够并行处理多个数据包，较大提高了入侵检测系统的工作效率，弥补了现有入侵检测系统因串行处理不及而丢失数据包的问题；

3、基于网络行为的分析技术

传统的入侵检测系统，只对存在威胁的行为进行分析，而一般的监控系统，则只对内部用户的行为进行分析，**AegisShield** 把两者结合起来，通过使用应用层传输数据特征库，快速识别出多达十余种中国内最常见的网络软件，如 QQ、P2P、HTTP 等；

4、基于 B/S 架构的 IDS

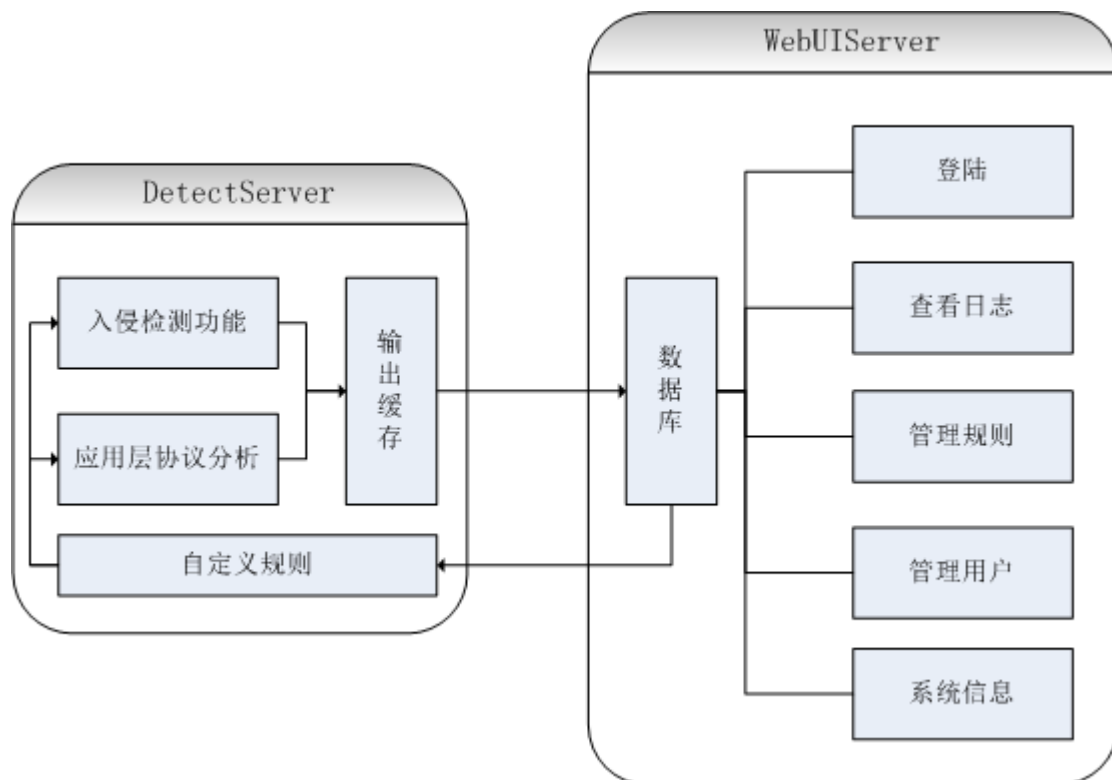
入侵检测系统作为一种网络防护机制，首先必须保证自身的安全。针对这一点，AegisShield 采用多种方式。对于系统的访问，采用分级机制，系统管理员和普通用户将具有不同的权限，同时，对 HTTP 协议包进行加密处理，确保用户的访问数据的安全性。

系统管理员应该有权限更改系统设置，主要是系统规则库的更改。在此我们将定义一套完整的规则设置语言，管理员可以使用这一套语言同时向几个引擎的规则库中添加、删除、更改规则，来设置网络的安全级别。如若管理员禁止对某一主机的访问，可将对此主机的所有访问设为非法等等。

作为一个 B/S 架构的入侵检测系统，所提供的界面必须简洁和易于操作，即界面必须友好。同时，系统需要为管理员或用户提供一个完整的网络状态分析，形式可以为饼图，直方图，曲线图等等，格式不局限。

另外，基于 B/S 的特点，让 AegisShield 具有跨平台特性，无论是使用 Linux 还是 Windows，甚至是在公交车上使用手机，也可以方便地进行 AegisShield 进行查看与操作。

2.3 主要功能概述



主要功能框架

AegisShield 主要分为 DetectServer 和 WebUIServer 两大部分。

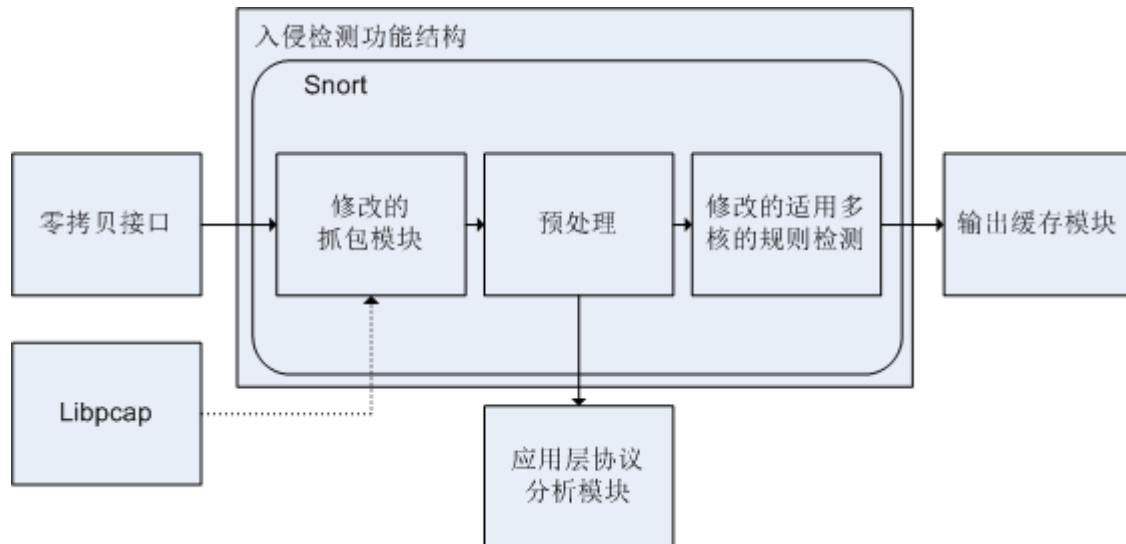
Detect Server

主要完成 Packet 抓取与分析过程，其处于一个较安全的网络当中，其中一个网络接口被设成只进行 Packet 抓取，另一网络接口与 WebUI Server 相连，传输数据。它提供的功能包括入侵检测分析、网络行为分析、自定义规则更新、缓存输出等。

WebUI Server

为用户提供 Web 接口，方便查看。同时与 Detect Server 和所在 Intranet 相连。为了确保 Detect 的安全性，避免 WebUI Server 受到攻击后 Detect Server 被入侵，它们中间由一个防火墙相连。它提供的功能包括 Web 登陆、查看日志、管理自定义规则、管理用户、查看系统信息等。

2.3.1 入侵检测功能



介绍：

1.AegisShield 实现了基于特征的入侵检测系统的基本功能。

2.由于修改了其数据包抓取接口，使用零拷贝接口作为输入（如上图所示），所有设置网络接口（如掩码、协议类型等）的功能参数将不可用。

3.由于零拷贝依赖于网络接口（网卡）驱动，为保证程序的兼容性，AegisShield 保留了 Libpcap 模块(虚线输入)，可以在编译时选择是否启用零拷贝功能。请使用以下格式关闭零拷贝功能：

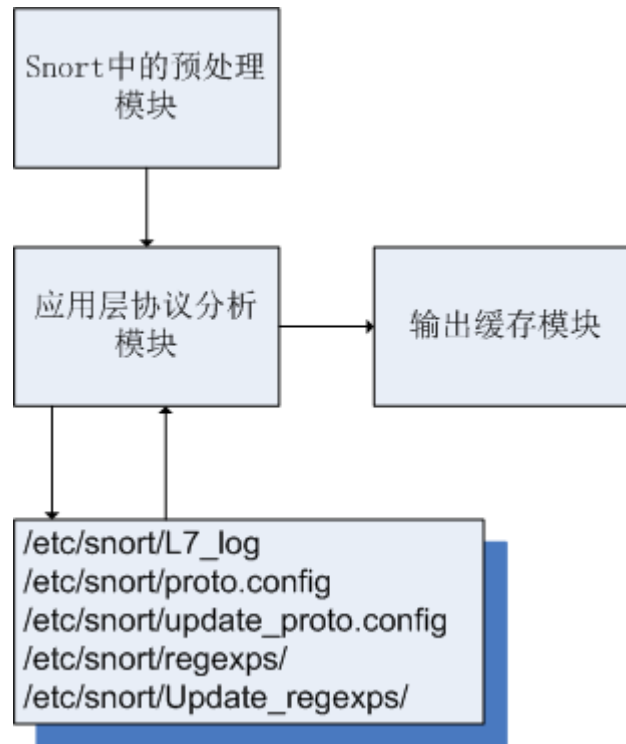
```
#define ZERO_COPY 0
```

4.各模块间使用流水线结构，可以充分使用多核，发挥其程序性能。由于模块间的不同步与并行，关闭了 snort 原有的 Profiling 功能。具体的内容请参考实现章节。

5.为了实现自定义规，在 snort 当中，etc/custom.rule 内容是可变的，由自定义规则模块负责初始化。请在 snort.conf 当中保证把该规则 include。

6.应用层协议分析模块依赖于预处理模块。二次开发时，必须注意。

2.3.2 应用层协议分析



介绍:

应用层协议分析模块是 AegisShield 下一个独立的分析模块，是基于数据流应用层内容的过滤。它使用模式匹配算法把进入设备的数据包应用层内容与事先定义好的协议规则进行比对，如果匹配成功就说明这个数据包属于某种协议。要注意的是，一个包只匹配一个规则，规则的先后顺序按/etc/proto.config 所示。proto.config 的内容格式如下：

第一行表示测检的网段；（如 192.168.1.0）

第二行表示掩码；（如 255.255.255.0）

第三行开始，表示协议名（对应 pat 文件）

因为一个数据流或者说一个连接的所有数据都是属于同一个应用的，所以没有必要对所有的数据包进行模式匹配，而只匹配一个流的前面几个数据包（AegisShield 系统定义为 5 个数据包）。当一个流的前面几个数据包包含了某种应用层协议的特征码时（比如 QQ），则这个数据流被识别；当前面几个数据包的内容没有包含某种应用层协议的特征码时，则放弃继续做模式匹配，这个数据流也就没有办法被识别。

匹配出的结果会输出缓存模块，最后输出到 `ag_rules` 数据库表，通过 WEB 服务器，管理员可以很轻松地添加、删除和修改自定义协议规则，在很大程度上减轻了管理员的负担。当数据库遭受攻击时，结果将保存到本地目录。即 `DetectServer` 的 `/etc/L7_log`。

在 AS 中，考虑到判断准确度、本地化等特点，默认选择的规则有以下：

编号	软件名称	测试版本	准确度	描述
1	QQ	Beta 1530	优	即时通讯软件
2	迷你迅雷	3.1.1.58	良	P2P 下载工具
3	电驴	1.1.7	良	P2P 下载工具
4	MSN	14.0.8089.726	优	即时通讯软件
5	FTP		优	文件传送协议
6	HTTP		优	超文本传输协议
7	SSH		优	Secure SHell
8	酷狗音乐	6.1.18.404	优	在线音乐播放器
9	DNS		优	域名解析协议
10	bittorrent	6.4 18095	良	BT 下载工具

更多的规则，请参考 `/etc/regexps/` 下的 `pat` 规则文件。该目录包括了所有可供使用的规则，文件名即为协议名。`pat` 文件的内容，以 QQ 通讯软件为例，主要内容如下：

#这是注释

qq

^.?.?\x02.+ \x03\$

非注释的第一行表示输出的协议名。非注释的第二行表示匹配的规则。使用特定规则的步骤如下：

1. 确定要进行检测的应用层协议（`regexps` 目录下的 `pat` 文件）
2. 把 `pat` 文件名（不包含 `pat`）添加到 `proto.config`
3. 重启 `snort`
4. 观察初始化信息，如果显示 `example.pat n success` 表示加载成功。

2.3.3 输出缓存

介绍:

AegisShield 使用双队列进行缓存输出:

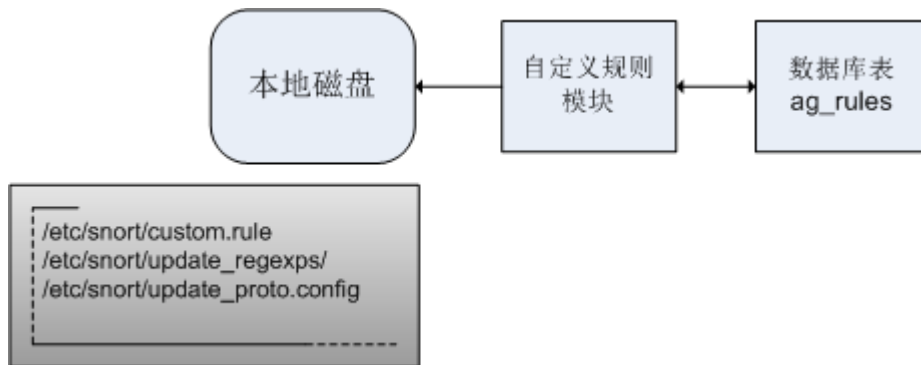
两个队列，一个给数据库输出线程读，一个给检测结果输入线程用来写，当数据库输出线程读完队列后会将自己的队列与检测结果输入线程的队列相调换。

检测结果输入线程每次写队列时都要加锁，数据库输出线程在调换队列时也需要加锁，但数据库输出线程在读队列时是不需要加锁。

队列缓冲区的大小是根据输入结果的数量的大小进行调整的，如果缓冲区很小，就能更及时的处理数据，但吞吐量以及出现资源竞争的几率大多了。由于输出到数据库优先率、重要性等相对其他模块不高，考虑到输出时，CPU 的 Cache 命中率，队列的实际大小与 Cache 大小相应。测试时，使用 1MB 的 L2 Cache，队列大小为 128KB，约为 128 个结果。

要注意的是，在超过上限的数量之后，将使用两个策略：1.直接丢弃。2.把输入挂载到延迟输入链表。考虑到现在的计算机主存足够大，**AegisShield** 使用第二种策略。

2.3.4 自定义规则



说明：

自定义规则的数据由 **WebUIServer** 提供。于 **DetectServer** 初始化时，将从远程获取自定义规则。这规则包括自定义规则和自定义应用层协议规则，出于安全性考虑，这些规则与预置的规则不重叠，**WebUI** 不能删除或修改原有的规则。产生的目录与文件如下所示：

/etc/custom.rule 保存自定义规则的文件

/etc/update_regexps/ 保存应用层协议分析自定义规则的目录

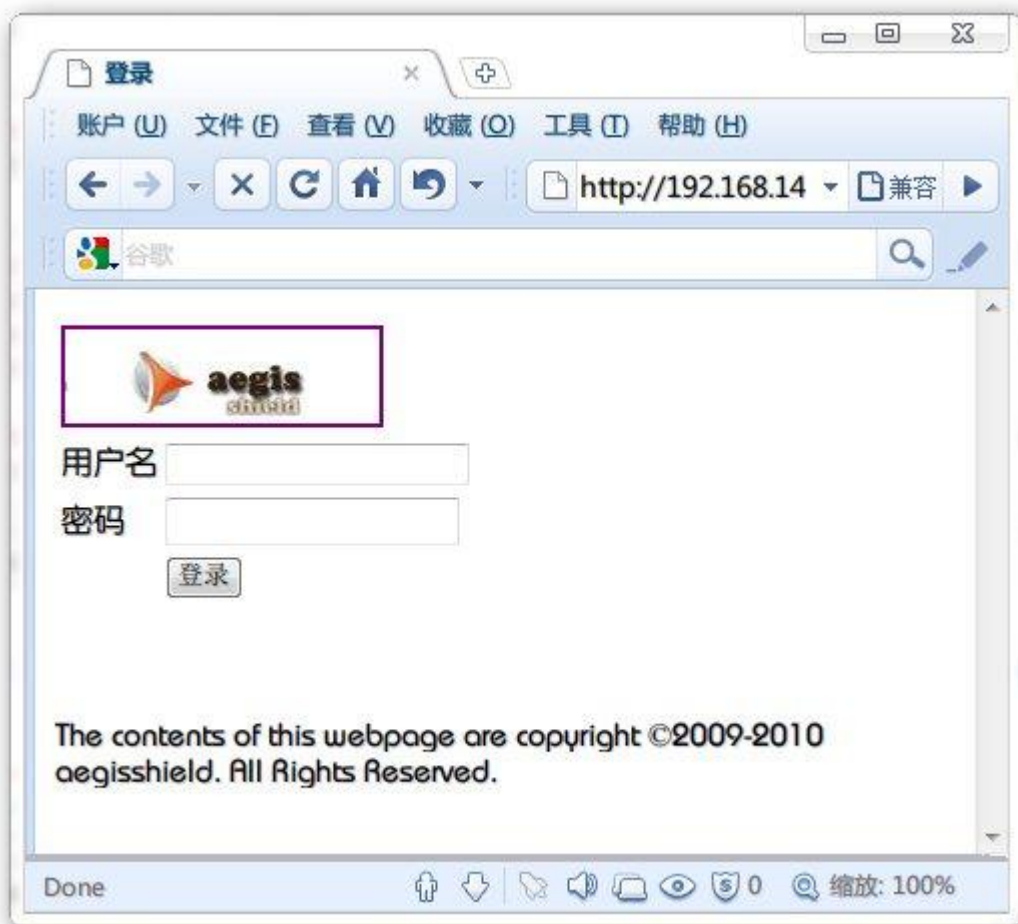
/etc/update_proto.config 保存应用层协议分析自定义规则列表

具体数据格式，请参考实现章节。

2.3.5 数据库

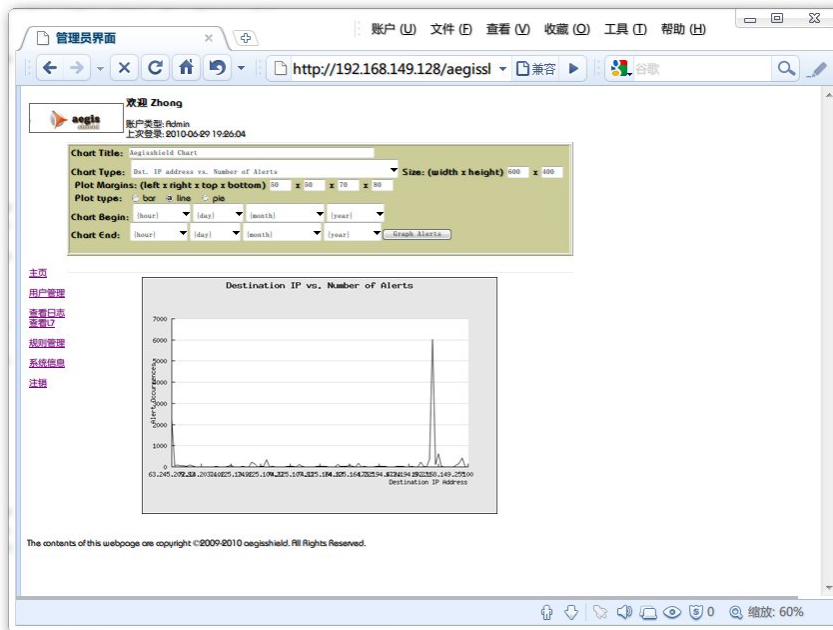
AegisShield 使用高效并免费的 MySQL 作为数据储存。并从安全角度考虑，Web 及其数据库设在了另一台服务器，数据库主要负责保存用户、日志事件、规则等。当数据库服务器遭受攻击时，AS 的 DetectServer 部分仍可以正常工作，并把信息记录到本地日志文件目录/etc/log/。

2.3.6 WEB 登陆



利用用户名与密码进行登陆，系统判断是否管理用户或一般用户。

2.3.7 查看日志



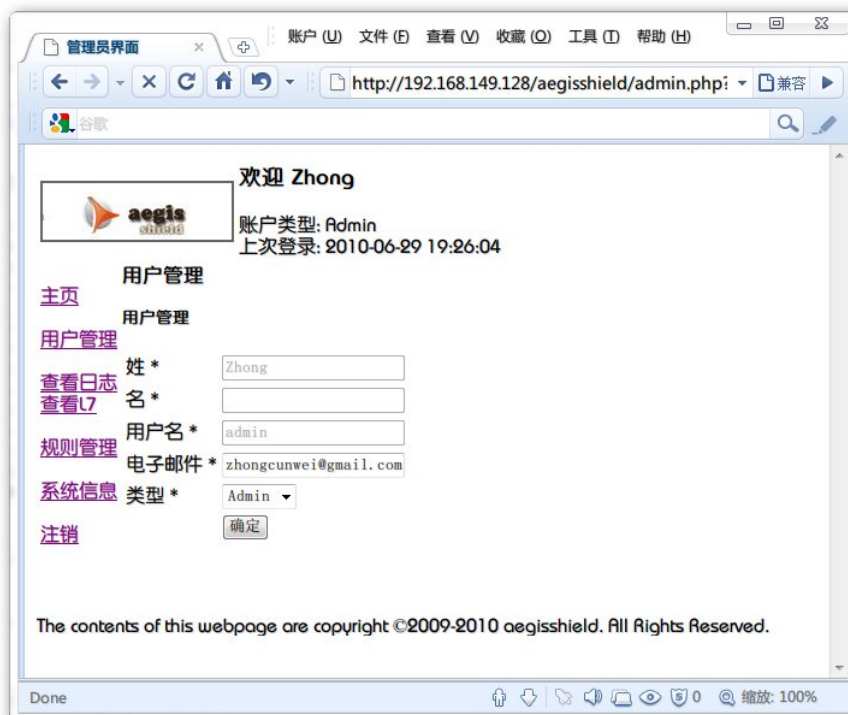
一般用户唯一的功能。可以查看 **DetectServer** 的输出。

2.3.8 管理规则



管理自定义规则，可添加删除修改。提供给 DetectServer 使用。

2.3.9 管理用户



管理不同权限的用户，用于登陆 Web，可添加删除修改。

2.3.10 系统信息



显示 WebUIServer 的基本信息。提供 SSH 接口，安全地对 DetectServer 操作。

2.4 概要实现方案

根据初始分析。针对各部分工作任务，主要工作的解决方案如下：

工作任务	解决方案
零拷贝实现	通过修改网卡驱动，把网络数据放到预先申请的缓存，并通过空间共享提供给接口调用。
Snort 零拷贝实现	主要修改 pcap 接口，把缓存中的数据交给下层处理。
CPU 调度优化	通过分析系统整体的性能，进行负载均衡，把任务绑定到 CPU。
多核优化实现	主要利用 Openmp 的并行指令，与 Intel 编译器进行优化。
应用层协议分析	根据不同应用层软件的数据包类型，定义相应的正则表达式。
WEB	通过分析 mysql 数据库表，利用 GD 库，分析与输出日志。
B/S 架构	通过 detect 与 web server 分离，分配不同的职能。

实现零拷贝任务

对不同型号的网卡驱动进行修改，把原先放到系统协议栈的数据，直接放在了预先分配的空间，减少数据拷贝和共享总线操作的次数，消除通信数据在存储器之间不必要的中间拷贝过程，降低系统调用的开销。

应用层协议分析工作

该模块用于对一个数据包的应用层协议做分析，来发现当前网络中使用的常用软件。首先，系统读取由用户指定的待检测协议，将用于协议匹配的正则表达式加载入内存。然后，当数据包到达时，系统将数据包中应用层数据匹配正则表达式，如果匹配成功，则输出源 IP、源端口、匹配的时间、协议名到数据库

多核优化工作——使用手动负载均衡

把已知的任务预先分配到指定的 CPU 上，对 CPU 进行绑定操作。在任务运行的时候，内核将不会对其任务进行负载均衡操作，保持该任务在指定 CPU 上运行。负

载均衡技术，将 CPU 的负荷平均分配到多个 CPU 核中，这就意味着，在比较繁忙的 CPU 核上运行的线程将可能会被操作系统自动迁移到空闲的 CPU 核上，这种迁移将导致被迁移的线程的 Context（包括寄存器值和 Cache 中的数据）需要迁移到新的 CPU 核上。如果这种迁移过于频繁，很显然会导致应用程序性能的下降。手动负载均衡将不会造成此情况，可以让 CPU 缓存保持热状态，并且不会发生任务转移。并充分发挥多核处理器，使用多线程和对不同数据包进行并行处理。

WEB 端工作

拟采用 php 进行开发，熟悉 snort 的输出格式，通过 GD 图形库进行直观的输出，提供友好简易的操作界面，同时利用 php 的 ssh 功能，提供 ssh 操作的相关接口。

概要项目实施进度

任务描述	任务开始时间	任务结束时间	提交物
1、项目启动及准备	2009/10	2009/11/30	无
2、开发环境与技术准备	2009/12	2010/2	工作月报
3、系统设计	2010/3	2010/5/15	概要设计初稿
4、详细设计与分析	2010/5	2010/6/1	详细设计初稿
5、设计与编码	2010/6	2010/6/15	各模块代码
6、集成与测试	2010/6	2010/6/30	测试文档
7、文档修订	2010/6	2010/6/20	概要、详细设计正稿

2.5 应用市场分析与后期开发

AegisShield2.0 基于多核处理平台的网络行为分析系统可以广泛的应用于大型电子商务、公司企业、学校等内部网络系统中。

对于大型的网络系统，网络流量大是一个主要的特点，一般 **IDS** 无法完成在高速网络下的检测任务，**AegisShield** 提供了以零拷贝为接口的 **IDS** 功能，安全可靠、易于管理的保护措施，对于内外网隔离的网络结构能够提供良好的支持。

对于公司企业网络，要求员工上班时间不运行其他不必要的网络软件，**AegisShield** 提供方便监控功能，管理员只需登陆 **Web** 端，即可查询到相关的信息。同时，**AegisShield** 的 **IDS** 功能，还能为企业提供一定安全。

对于学校网络，要求学生规范上网，应用层协议分析功能可以把特定的网站 **URL** 加入到规则当中，由于 **http** 也是应用层协议的一种，**AegisShield** 系统自然可以对其进行分析。

从商业角度来说，**AegisShield** 还存在一些不足，如不能自动更新规则，必须进行手动添加，零拷贝的稳定性还不是很强等。而且安全性也有待提高，由于 **Web** 端与 **Detect** 端相连，这必然是一个隐患。**DetectServer** 以开最小服务以获得安全性的原理，只开放了 **ssh** 服务。

以下应用是后期开发当中，应该完善的：

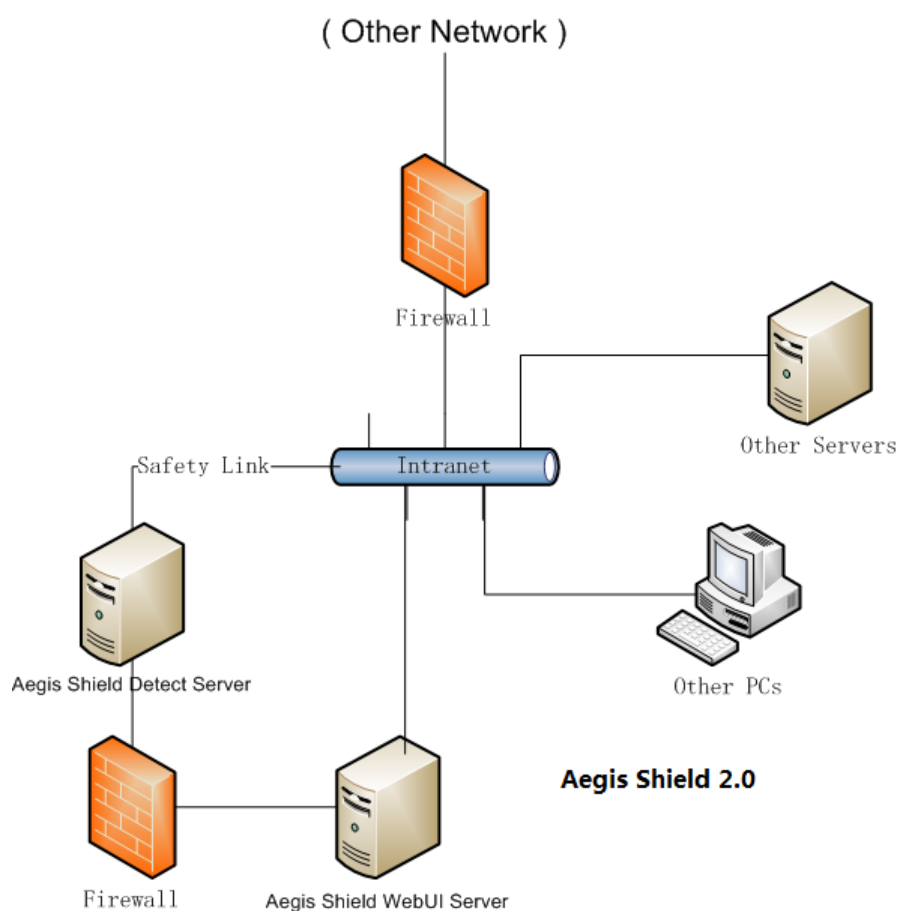
- 1.改进系统匹配算法，提高系统处理效率；
- 2.改善零拷贝稳定性不好的问题；
- 3.改进多核算法，进行更大程度的优化；
- 4.增强用户行为分析能力，增加匹配的行为数目，同时保证检测的准确率；
- 5.双队列的缓存性能还有待提高，使用循环缓存更能提升性能；
- 5.可自定义加载检测模块，针对不同的用途；

6.重新编写 **web** 页面，使其更美观，更加人性化，同时让 **web** 的功能更为丰富，以方便管理员使用。

第三章 实现方案

3.1 总体设计

3.1.1 系统构架



AegisShield 软件构架

系统主要由四部分组成，按照数据流的方向，其分别是：

Safety Link：通过 Intranet 的交换机或路由配置，把数据包转发至该端口。

AegisShield DetectServer：检测服务器。包括以 snort 为核心的检测引擎。

中间 **Firewall**：与 WebUIServer 相连，保证安全性。

AegisShield WebUIServer：提供 User Interface。即 Web 功能。

3.1.2 技术路线

要求环境	内容
操作系统	Linux 2.6.18 以上
内存	1GB RAM
支持网卡	Pcnet32 igb, for Intel 82575 ixgbe, for Intel 82598 bnx2, for BCM Server GE sky2,Marvell serises. 注：要求有两块不同型号的网卡，至少有一块为上述型号。
Snort 开发	使用 C，Mysql-devel
Web 端开发	Apache2，php5.2 以上，GD2.0 以上
其他	Intel C++ Compiler 9.0

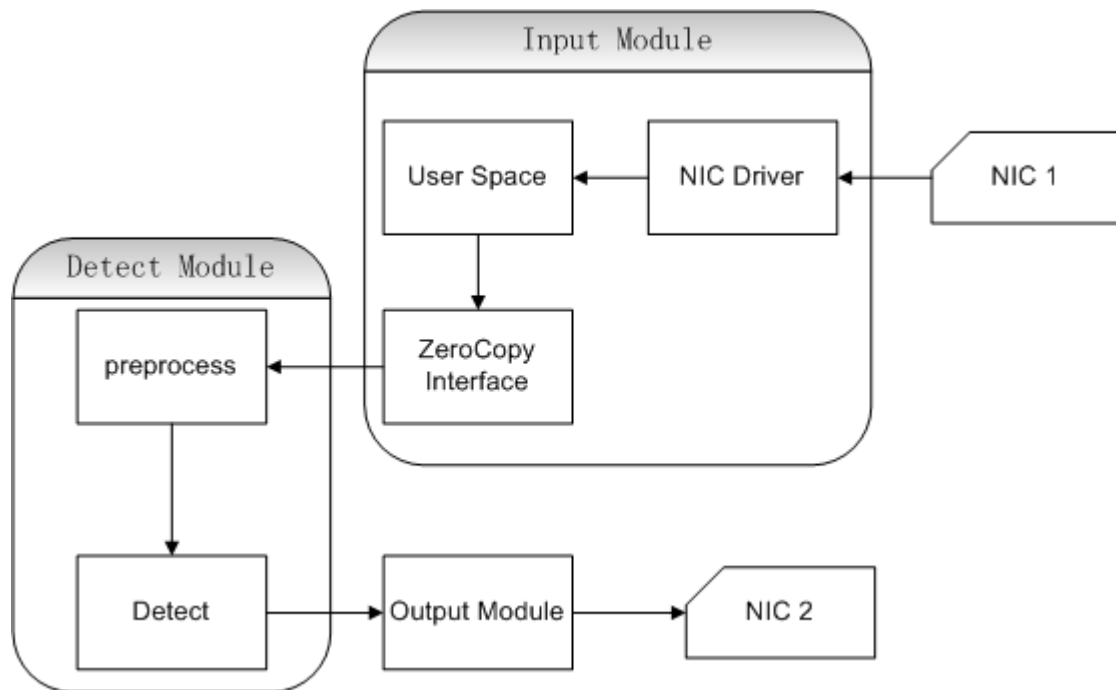
说明：

由于零拷贝模块在初始化时须占用大量的内核内存，以便数据包不被丢弃，在保证系统的性能与稳定性，推荐使用更高的内存。

零拷贝是依赖网络接口（网卡）驱动的，因此仅此支持部分驱动。

考虑到兼容性，Web 端推荐使用 php5.2 + GD2.0(须为内置版本)，GD 用于图形化输出检测结果。

3.1.3 模块结构



主要分成四大模：

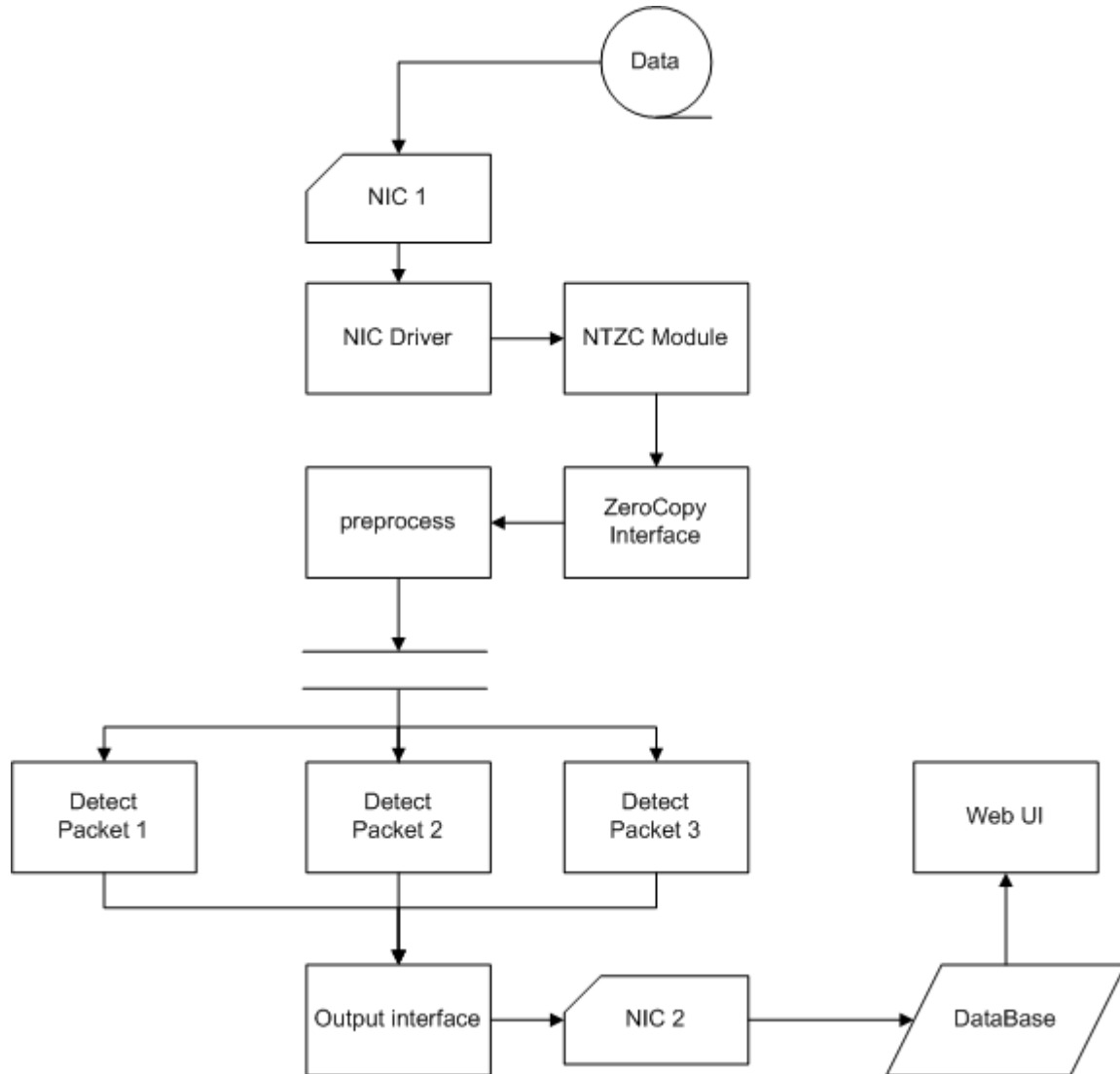
Input：零拷贝的主要实现，其主要包括 NIC 驱动、自定用户级共享空间管理、零拷贝数据包截取等子模块。

Detect：包括预处理和检测等两个子模块，检测子模块分为入侵检测与应用层协议检测两个模。

Ouput：包括缓存输出、输出到到 Mysql 与本地日志保存等模块。

Web：包括用户管理、规则管理、日志查看等三个模块。（上图未标出）

3.1.4 处理流程



AegisShield 2.0 总体流程图

主要的数据流有：

- ①NIC 1； ②NTZC Module； ③preprocess；
④并行 Detect； ⑤Output； ⑥NIC 2； ⑦WebUI

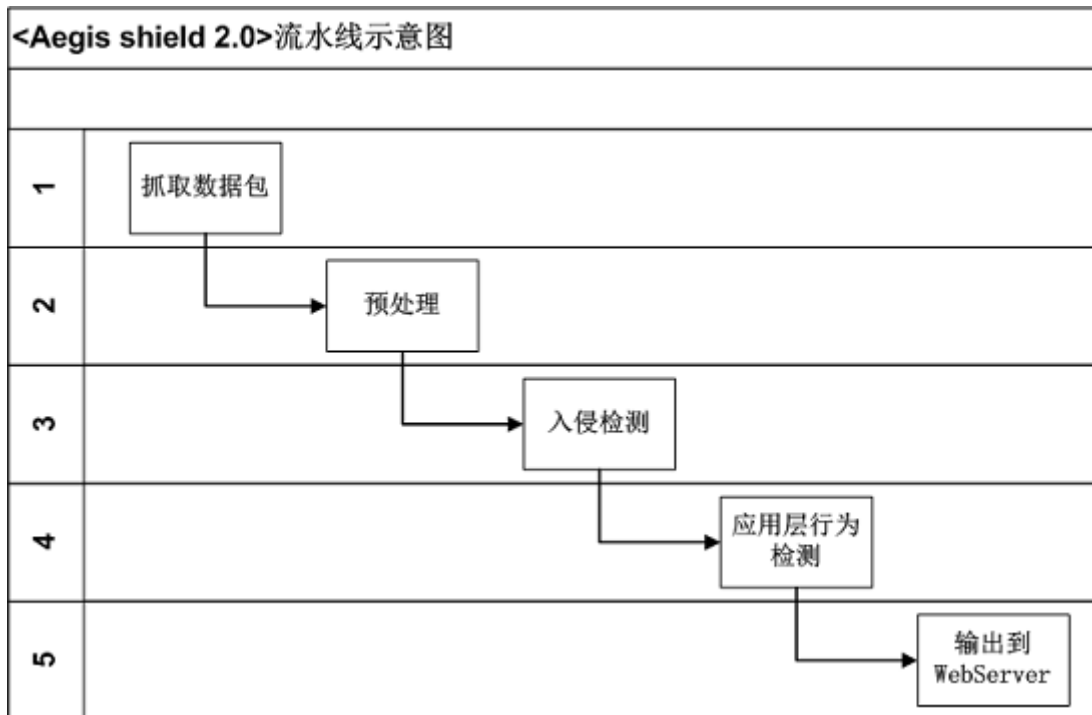
数据流的各部分内容，见下表。

AegisShield2.0 数据流内容

顺序号	内容
①	数据从 NIC1 流入
②	通过修改的网卡驱动，映射到内核空间， ntzc 提供内存管理
③	预处理器利用零拷贝接口取得数据
④	分派给多线程并行处理数据包
⑤	格式化输出、缓存结果
⑥	通过 NIC2 输出到 mysql
⑦	Web UI 读取 mysql ，显示给用户查看

3.1.5 流水线规划

AegisShield2.0 使用流水线来划分系统的可并行部分，从简入深，从串行开始逐步介绍 AegisShield2.0 的流水线并行方法与策略，在串行执行中，其流水线如下图：



说明：

AegisShield 总体的流水线节点分解(Part1 – Part5)如上图，图中并没有直接标明节点的并行优化。其中，Part3 与 Part4 间不存在依赖相关，可以直接并行执行。其他流水线节点，通过生产者/消费者的模型来实现，上层为下层提供所必要的的数据。由于采用了流水线的模式，其整体效率由瓶颈的效率决定：

整体的效率 $P = \text{Max} (\text{Part1} , \text{Part2} , \text{Part3} , \text{Part4} , \text{Part5})$

3.1.6 数据库设计

```
/*=====*/  
/* Table: 应用层协议分析结果 */  
/*=====*/
```

```
create table appproto  
(  
    id          int    unsigned not null    auto_increment,  
    time        int    unsigned not null ,  
    ip          int    unsigned not null,  
    port        smallint unsigned not null,  
    proto       varchar(255) not null,  
    primary key (id)  
);
```

表项	作用
id	标识该事件
time	事件发生的时间
ip	产生事件的主机网络地址
port	产生事件的主机网络地址端口
proto	说明事件的活动描述


```

/*=====*/
/* Table: 登陆信息表                                */
/*=====*/

```

```

CREATE TABLE IF NOT EXISTS `ag_ip_history` (
  `user_id` int(10) NOT NULL,
  `date` datetime NOT NULL,
  `state` int(3) NOT NULL,
  `ip` char(15) NOT NULL,
  PRIMARY KEY (`user_id`,`date`,`state`)
) ENGINE=MyISAM DEFAULT CHARSET=gb2312;

```

表项	作用
user_id	用户 ID
date	session 发生的时间
state	当前用户状态
ip	操作的 IP

```

/*=====*/
/* Table: 自定义规则表                                */
/*=====*/

```

```

CREATE TABLE IF NOT EXISTS `ag_rules` (
  `id` int(10) NOT NULL,
  `type` int(11) NOT NULL,
  `rule` mediumtext NOT NULL,
  `proto` int(11) NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=gb2312;

```

表项	作用
id	标识自定义规则
type	规则类型
rule	规则字符串
proto	规则描述

```

/*=====*/
/* Table: Session 会话记录表 */
/*=====*/
CREATE TABLE IF NOT EXISTS `ag_sessions` (
  `id` varchar(32) NOT NULL,
  `user_id` int(10) NOT NULL,
  `vars` mediumtext,
  `date` datetime NOT NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=gb2312;

```

表项	作用
id	标识 session
user_id	用户 ID
vars	环境变量，用于保存设置
date	活动时间

```

/*=====*/
/* Table: 用户表 */
/*=====*/
CREATE TABLE IF NOT EXISTS `ag_users` (
  `id` int(10) NOT NULL auto_increment,
  `username` varchar(20) NOT NULL,
  `passwd` varchar(100) NOT NULL,
  `email` varchar(100) NOT NULL,
  `language` varchar(255) NOT NULL,
  `created` datetime default NULL,
  PRIMARY KEY (`id`)
) ENGINE=MyISAM DEFAULT CHARSET=gb2312 AUTO_INCREMENT=3 ;

```

表项	作用
id	标识用户
username	用户名
passwd	密码
email	用户 email
language	界面语言
create	创建时间

3.2 模块概要设计

3.2.1 零拷贝模块

模块名称	零拷贝模块		父模块	无	标识	ZC00
接口说明	输入	网卡				
	输出	给本行为分析系统中央处理模块				
功能说明	零拷贝模块通过提供自定义的缓冲区，将网卡嗅探到的数据传送给此缓冲区，后行为分析模块从缓冲区中取得数据，进行分析					
支持环境	linux，内核版本 2.6 以上。					
依赖关系	调用模块	五个子模块：网卡驱动模块，驱动接口模块，内存管理模块，用户接口模块，用户接口				
	被调用模块	行为分析模块				

模块名称	网卡驱动模块		父模块	零拷贝模块	标识	ZC01
接口说明	输入	网卡				
	输出	内存管理模块				
功能说明	此模块为修改过的支持零拷贝的网卡驱动，数据包网卡接收到之后，并不会存放到系统默认的协议栈缓冲区，而是转移到 ntzc 的自定义缓冲区。这种功能的实现必然要依靠网卡驱动的支持。					
支持环境	根据内核版本和网卡的不同，需要使用不同的网卡驱动，关于支持的网卡驱动，需要进行定制。					
依赖关系	调用模块	驱动接口模块				
	被调用模块	网卡				

模块名称	驱动接口模块		父模块	零拷贝模块	标识	ZC02
接口说明	输入	无				
	输出	无				
功能说明	此模块主要为网卡驱动提供必要的零拷贝接口，是连接网卡和自定义缓冲区的桥梁。网卡驱动通过使用此模块的接口，将网卡数据包转移到 ntzc 的缓冲区。					
支持环境	linux，内核 2.6 以上					
依赖关系	调用模块	内存管理模块				
	被调用模块	网卡驱动模块				

模块名称	内存管理模块		父模块	零拷贝模块	标识	ZC03
接口说明	输入	将数据包存放到缓冲区				
	输出	将数据包提取出来供协议分析使用				
功能说明	内存管理模块是整个零拷贝模块的核心模块，负责内核自定义缓冲区的分配，管理，释放；整个零拷贝模块即使围绕这一子模块完成数据包的整个流动。					
支持环境	linux，内核版本 2.6 以上					
依赖关系	调用模块	无				
	被调用模块	用户接口模块，驱动接口模块				

模块名称	用户接口模块		父模块	零拷贝模块	标识	ZC04
接口说明	输入	从自定义缓冲区取出数据包				
	输出	将数据包传送给协议分析模块				
功能说明	为内核缓冲区与用户的交互提供管理，将自定义缓冲区设置为虚拟设备(/dev/zc)缓冲区，用户可以通过映射（MMAP）此缓冲区来达到最网络数据包的访问。					
支持环境	linux，内核版本 2.6 以上					
依赖关系	调用模块	内存管理模块				
	被调用模块	用户接口				

模块名称	用户接口	父模块	零拷贝模块	标识	ZC05
接口说明	输入	自定义缓冲区，表现为设备缓冲区			
	输出	数据包			
功能说明	此模块名称为 用户接口，须注意此模块与“用户接口模块”不同，勿混淆。用户接口模块将内核自定义缓冲区设置为虚拟设备（/dev/zc）缓冲区，而用户接口则提供了一系列用户程序可直接使用的接口，在我们的系统中提供了三个类似 libpcap 的抓包接口，即存放在此模块中				
支持环境	linux，内核 2.6 以上				
依赖关系	调用模块	用户接口模块			
	被调用模块	用户程序			

3.2.2 应用层协议分析模块

模块名称	应用层协议分析模块		父模块	无	标识	LA00
接口说明	输入	以太网数据包				
	输出	源 IP、源端口、协议名、时间				
功能说明	该模块用于对一个数据包的应用层协议做分析，来发现当前网络中使用的常用软件。首先，系统读取由用户指定的待检测协议，将用于协议匹配的正则表达式加载入内存。然后，当数据包到达时，系统将数据包中应用层数据匹配正则表达式，如果匹配成功，则输出源 IP、源端口、匹配的时间、协议名到数据库					
支持环境	Mysql: 分析出的结果通过 mysql 输出 Snort: snort 提供的以太网数据包作为输入					
依赖关系	调用模块	规则文件管理模块				
		连接管理模块				
规则匹配模块						
输出过滤模块						
输出模块						
	被调用模块	Snort 检测模块				

模块名称	规则文件管理模块		父模块	应用层协议分析模块	标识	LA01
接口说明	输入	本地规则配置文件、WEB 端规则配置文件				
	输出	规则正则表达式链表				
功能说明	1. 读取本地规则配置文件。 2. 根据配置文件查找指定协议的规则文件。 3. 将规则文件中的正则表达式编译并加载入内存 4. 读取 WEB 端规则配置文件 5. 根据配置文件查找指定协议的规则文件。 6. 将规则文件中的正则表达式编译并加载入内存					
支持环境	本地规则配置文件：由管理员在本地指定的需要检测的应用层协议名 WEB 端规则配置文件：由管理员在 WEB 端指定的需要检测的应用层协议名 本地规则文件夹：由管理员在本地指定的需要检测的规则文件集 WEB 端规则文件夹：由管理员在 WEB 端指定的需要检测的规则文件集，其通过数据库来获取					
依赖	调用模块	无				
关系	被调用模块	应用层协议分析模块				

模块名称	连接管理模块		父模块	应用层协议分析模块	标识	LA02
接口说明	输入	以太网数据包				
	输出	源 IP、源端口、目的 IP、目的端口、协议名、协议号、时间				
功能说明	1. 查找当前连接（由源 IP,源端口号,目的 IP,目的端口号组成）是否是旧连接，若是，则读取旧连接，否则新建一个连接。 2. 查看该连接是否已经匹配成功,若匹配成功,则跳过后续操作 3. 若该连接上数据未匹配成功过,则遍历正则表达式链表，匹配当前连接上应用层的数据 4. 若匹配成功，则输出源 IP、源端口号、协议名、匹配成功的时间 5. 若连续匹配失败超过 5 次，则删除该连接					
支持环境	Snort: snort 提供的以太网数据包作为输入					
依赖	调用模块	规则匹配模块				
关系	被调用模块	应用层协议分析模块				

模块名称	规则匹配模块	父模块	连接管理模块	标识	LA03
接口说明	输入	经过连接过滤的以太网数据包			
	输出	分析出的协议号、协议名			
功能说明	1. 去除应用层数据中的 0x00，防止匹配不完整 2. 遍历正则表达式链表，逐个匹配正则表达式 3. 若匹配成功，则返回协议号和协议名称 4. 若匹配不成功，返回协议号为 0				
支持环境	无				
依赖	调用模块	无			
关系	被调用模块	连接管理模块			

模块名称	输出过滤模块		父模块	应用层协议分析模块	标识	LA04
接口说明	输入	由规则匹配模块输出的协议号、协议名 由连接管理模块输出的源 IP				
	输出	过滤的结果（源 IP，源端口、协议名、匹配时间）				
功能说明	检查当前检测到的协议是否已经检查过 若检查过，并且超过了检查间隙时间，则更新检查时间；反之，跳过后续操作 若没检查过，则将记录下当前协议名和匹配时间等数据					
支持环境	无					
依赖	调用模块	无				
关系	被调用模块	应用层协议分析模块				

模块名称	输出模块	父模块	应用层协议分析模块	标识	LA05
接口说明	输入	输出过滤模块过滤后的匹配结果（源 IP，源端口、协议名、匹配时间）			
	输出	数据库操作语句字符串			
功能说明	写线程将数据拷贝到由写指针指定的缓冲区 读线程将数据从由读指针指定的缓冲区读取数据，并执行数据库 IO 操作。				
支持环境	Mysql: 分析出的结果通过 mysql 输出				
依赖	调用模块	无			
关系	被调用模块	应用层协议分析模块			

3.2.3 多核优化模块

模块名称	静态 CPU 调度模块		父模块	无	标识	MC00
接口说明	输入	流水线中对应的节点代号，如预处理、入侵检测等				
	输出	CPU 静态调度方案				
功能说明	根据不同的 CPU 数目，利用已定义的静态均衡算法，对进程进行 CPU 绑定，以优化程序性能。					
支持环境	要求 Intel C++ Compiler 9.0 提供支持，利用 Openmp 接口进行亲和力设置					
依赖关系	调用模块	无				
	被调用模块	预处理模块（snort）、入侵检测模块（snort）、应用层协议分析模块				

3.2.4 WEB 模块

模块名称	数据库配置连接模块		父模块	无	标识	WB00
接口说明	输入	数据库信息配置文件				
	输出	数据库连接状态				
功能说明	1. 读取数据库配置文件。 2. 根据数据库配置文件中的信息连接数据。 3. 成功连接将保存与数据库连接的会话					
支持环境	无					
依赖关系	调用模块	无				
	被调用模块	登录模块				

模块名称	登录模块		父模块	无	标识	WB01
接口说明	输入	用户名，密码				
	输出	登录后的操作界面				
功能说明	该模块用于判断用户登录，如果登录成功，将进入用户操作页面；如果登录失败，将提醒一个错误信息。					
支持环境	Mysql: 判断用户名密码是否正确 浏览器: 显示操作后的结果					
依赖	调用模块	数据库连接配置模块				
关系	被调用模块	无				

模块名称	用户管理模块		父模块	登录模块	标识	WB02
接口说明	输入	无				
	输出	用户的基本信息				
功能说明	1.从数据库中获得用户组 2.根据不同类型显示所有用户 3.查看某个用户的基本信息 4.修个某个用户的密码 5.删除一个用户 6.查看用户的登录情况 7.添加一个用户					
支持环境	账户具有管理员权限					
依赖关系	调用模块	无				
	被调用模块	登录模块				

模块名称	日志管理模块		父模块	登录模块	标识	WB03
接口说明	输入	时间，图表类型参数				
	输出	警报数、柱形图、饼图、line 图				
功能说明	显示 snort 的基本情况，如警报数等 用柱形图来显示时间与警报数的关系 用饼图来显示时间与警报数的关系 用 line 图来显示时间与警报数的关系					
支持环境	管理员账号和普通账户					

依赖 关系	调用模块	无
	被调用模块	登录模块

模块名称	用户层协议信息模块		父模块	登录模块	标识	WB04
接口说明	输入	无				
	输出	数据库中检测到的协议基本信息图形表示				
功能说明	显示协议的基本信息					
支持环境	管理员账户、普通账户					
依赖关系	调用模块	无				
	被调用模块	登录模块				

模块名称	规则管理模块	父模块	登录模块	标识	WB05
接口说明	输入	自定义规则			
	输出	数据库中的规则			
功能说明	用户在添加规则项中填写规则 通过数据库连接将规则存入 显示数据库中的所有规则 修个某一条数据库中的规则 删除某一个数据库中的规则				
支持环境	Mysql: 规则显示通过 mysql 输出 管理员账号和普通账户				
依赖	调用模块	无			
关系	被调用模块	登录模块			

模块名称	系统信息模块		父模块	登录模块	标识	WB06
接口说明	输入	无				
	输出	当前平台的基本信息				
功能说明	显示当前操作系统基本信息，操作系统版本，数据库版本等 显示数据中表单情况					
支持环境	Mysql: 规则显示通过 mysql 输出 管理员账号和普通账户					
依赖 关系	调用模块	无				
	被调用模块	登录模块				

模块名称	注销模块		父模块	登录模块	标识	WB07
接口说明	输入	无				
	输出	无				
功能说明	用户注销登录					
支持环境	管理员账号和普通账户					
依赖关系	调用模块	无				
	被调用模块	登录模块				

3.3 模块详细设计

3.3.1 零拷贝模块

模块名称	零拷贝模块（NTZC--Network Tapping Zero Copy)	标识	ZC00
父模块	无	回溯标识	ZC00
模块功能	零拷贝模块通过提供自定义的缓冲区，将网卡嗅探到的数据传送给此缓冲区，后行为分析模块从缓冲区中取得数据，进行分析		

<p>相关数据</p>	<p>这里多数重要数据结构都存于文件 <code>zc_comm.h</code> 中，一些重要的结构题如下：</p> <pre> struct zc_data { union { __u32 data[2]; void *ptr; } data; __u32 off; __u16 r_size; /* the packet size */ __u8 entry; __u8 cpu:4, netdev_index:4; }; struct zc_ring { __u16 zc_pos; __u16 zc_used; }; struct zc_ring_ctl{ __u16 zc_used /* eraser */, zc_pos /* producer */; __u16 zc_prev_used, zc_dummy /* consumer */; }; struct zc_sniff { int sniff_id; int dev_index; int sniff_mode; #define ZC_SNIFF_NONE 0 #define ZC_SNIFF_RX 1 #define ZC_SNIFF_TX 2 #define ZC_SNIFF_ALL 3 u_int16_t pre_p; #define ZC_PRE_P_NPCP 0x8050 #define ZC_PRE_P_NORMAL 0 u_int16_t pre_type; </pre>
-------------	--

其中最重要一个的数据结构为 m_buf,定义如下:

```
struct m_buf {
    /* These two members must be first. */
    struct m_buf      *next;
    struct m_buf      *prev;

    struct sock       *sk;
    //ktime_t          tstamp;
    struct net_device *dev;

    /*
     * This is the control buffer. It is free to use for every
     * layer. Please put your private variables there. If you
     * want to keep them across layers you have to do a
     mbuf_clone()
     * first. This is owned by whoever has the mbuf queued
     ATM.
     */
    char              cb[48];

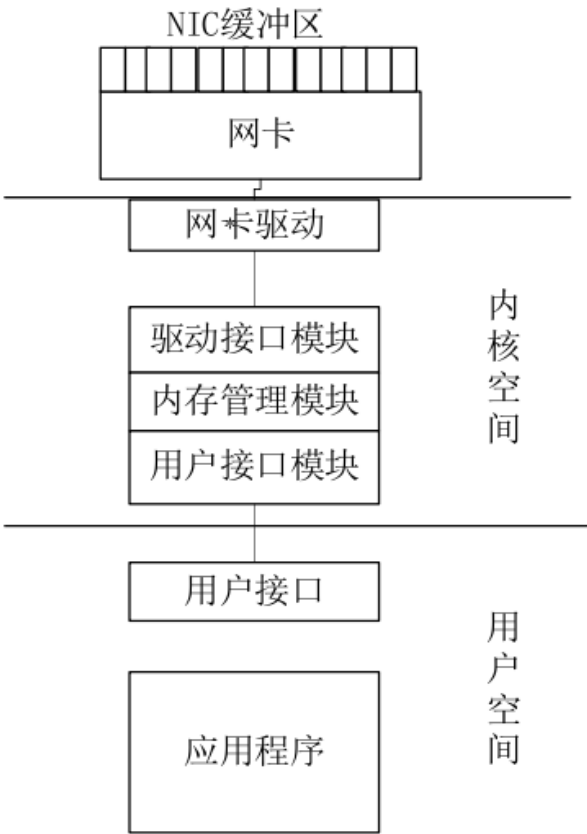
    unsigned int      len,
                     data_len;
    __u16             mac_len,
                     hdr_len;
    union {
        __wsum        csum;
        struct {
            __u16      csum_start;
            __u16      csum_offset;
        };
    };
    __u32             priority;
    __u8              local_df:1,
                     cloned:1, 第 46 页
                     ip_summed:2,
                     nohdr:1,
```

	<pre> m_buf_data_t transport_header; m_buf_data_t network_header; m_buf_data_t mac_header; /* These elements must be at the end, see alloc_mbuf() for details. */ m_buf_data_t tail; m_buf_data_t end; unsigned char *head, *data; unsigned int truesize; atomic_t users; }; </pre> <p>这个结构体定义于 <code>nta.h</code>, 本质上只是原系统 <code>sk_buff</code> 的拷贝, 对比其中的数据对象即可发现。数据包到达后将填充结构体 <code>m_buf</code>, 而非原先系统的 <code>sk_buff</code>。故这个结构体对系统具有及其重要的意义。</p>
限制条件	<p>本系统由于需要自己定义内核缓冲区, 因而需要占用大量系统内存, 很容易造成系统内存不足, 这是一个问题, 但是牺牲空间换时间, 以此达到提高性能的目的, 我认为们是值得的。</p> <p>此模块总体提供了三个接口, 类似与 <code>libpcap</code>, 用户可以以类似 <code>libpcap</code> 的使用方法来获取数据包。</p>
输入数据	到达网卡的数据包
输出数据	类似于 <code>libpcap</code> 的接口, 用户可以调用 <code>zc_loop</code> 对获得的数据包进行处理

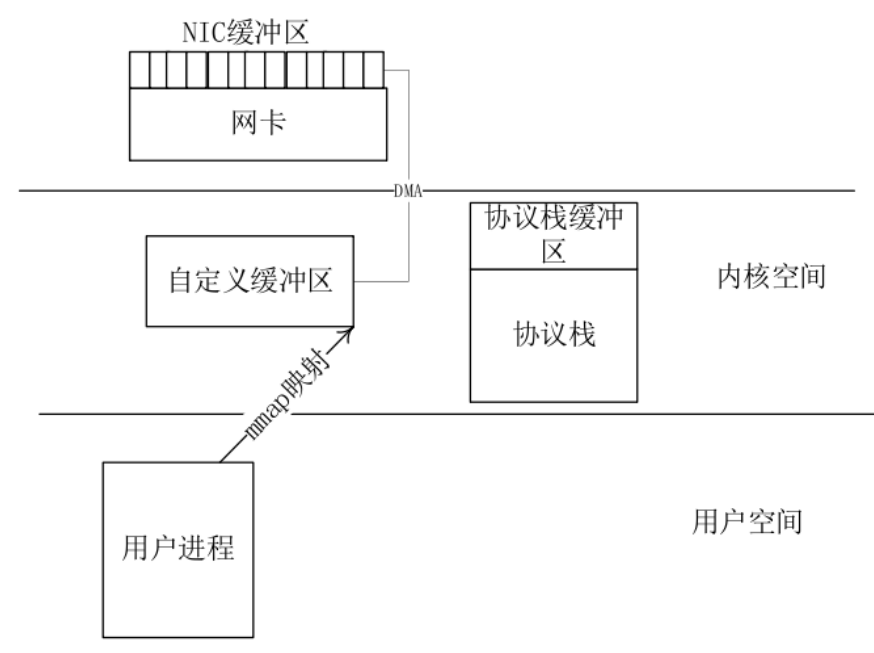
<p>图例</p>	<p>首先说明一下零拷贝模块的总体代码结构（这里将提供的更改过的网卡驱动去掉，方便说明）：</p> <pre>ntzc -- nta -- Makefile -- control.c -- control.h -- send.c -- sniff.c -- zc -- Makefile -- bvl.c -- bvl.h -- nta.c -- nta.h -- zc.c '-- zc_comm.h</pre> <p>各个文件所属模块及简要说明：</p>

文件	所在文件夹	所属模块	功能
Bvl.h bvl.c	Zc	内存管理模块	Ntzc 的核心模块，为用户自定义的内存缓冲区提供管理，包括分配，回收，计数等等功能
Nta.h Nta.c	Zc	驱动接口模块	为网卡驱动提供了内存管理接口，网卡驱动将利用这些接口将数据包转移到自定义的缓冲区，而忽略协议栈缓冲区
Zc.c Zc_comm.h	Zc	用户接口模块	为内核缓冲区与用户的交互提供管理
Control.h Control.c	Nta	用户接口	为用户提供零拷贝接口，用户将利用零拷贝接口完成自己的功能
Send.c Sniff.c	nta	应用程序	零拷贝使用实例应用程序，可以不研究

而零拷贝各个子模块的结构设计图如下：

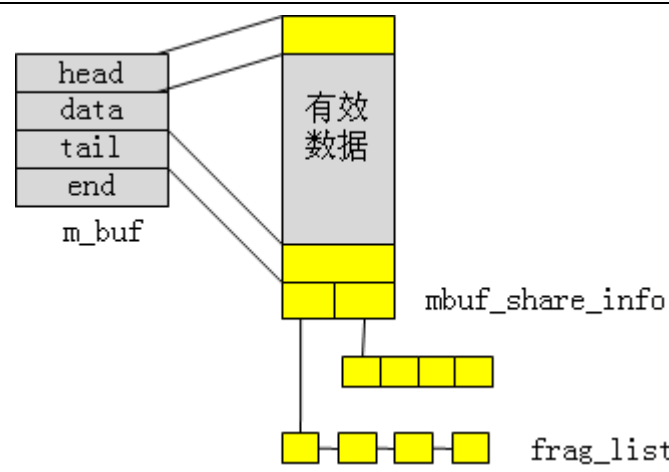


算法逻辑	后面将详述。
调用接口	无
提供接口	<pre> zc_t* zc_open_live(const char*dev, int snaplen, char* errbuf); void zc_destroy(zc_t* zc_ctl); int zc_loop(zc_t *zc_ctl, int cnt, void (*zc_handler)(unsigned char* user, const struct pcap_pkthdr *h, const unsigned char *bytes), unsigned char* user); </pre> <p>这三个函数是仿照 <code>libpcap</code> 的函数原型编制而成，用户可参考 <code>pcap_open_live</code>, <code>zc_destroy</code>, <code>zc_loop</code> 的用法使用此三个接口。</p>

<p>备注</p>	<p>由于零拷贝模块为父模块，这里要说明一下零拷贝的整体设计思路：</p>  <p>The diagram illustrates the zero-copy architecture across three memory spaces: User Space (用户空间), Kernel Space (内核空间), and NIC Buffer (NIC缓冲区). In the User Space, a 'User Process' (用户进程) is shown. In the Kernel Space, there is a 'Custom Buffer' (自定义缓冲区), a 'Protocol Stack Buffer' (协议栈缓冲区), and the 'Protocol Stack' (协议栈). The NIC Buffer and 'NIC' (网卡) are located in the hardware layer. Data flow is as follows: 1. A 'DMA' arrow points from the NIC Buffer to the Custom Buffer. 2. An 'mmap映射' arrow points from the User Process to the Custom Buffer. 3. A dashed arrow points from the Protocol Stack Buffer to the Custom Buffer, indicating that the protocol stack is bypassed. 4. A dashed arrow points from the Custom Buffer to the NIC, indicating that data is sent directly to the network card without being copied into the protocol stack buffers.</p> <p>Ntzc 通过修改过的驱动将数据包劫持到自定义的缓冲区 (DMA 传输) ,用户进程 mmap 到此内存区域,从而达到 零拷贝的目的,在此过程中,显而易见,网卡数据将无视内核协议栈缓冲区的存在,TCP/IP 协议栈完全作废,故使用零拷贝期间,无法正常上网。</p> <p>NTZC 的设计中,认为既然用户空间可以以足够低的成本收发报文了,因此取消了网卡驱动和 Linux 协议栈之间的交互,当然,从 NTZC 管理的报文内存中拷贝一份出来交给标准协议栈处理,也是很容易的事情。</p> <p>由于优先考虑到不对内核打补丁,因此没有修改 sk_buff 的内存管理机制,相反的,实现了和 sk_buff 接口语义几乎相同,但内存管理机制发生变化的报文数据结构(m_buf,只是借用了 BSD 里面的数据结构名称,骨子里就是对 sk_buff 的复制) 。使用这个 m_buf 数据 结构及 API,需要对网卡驱动进行修改。</p>
-----------	--

	<p>正因为是个通用的零拷贝支持模块,网卡驱动的修改很容易,全局替换 <code>sk_buff</code> 的操作到 <code>m_buf</code> 的操作即可,因此,理论上任何在 <code>Linux</code> 中已经有源代码支持的 <code>NIC Driver</code>,都很容易被改造成 <code>NTZC</code> 的 <code>NIC</code> 驱动,只是,这块网卡将不能在被 <code>Linux</code> 协议栈使用。<code>NTZC</code> 的代码中也给出了 <code>Intel 82575</code> 改造后的驱动作为例子。</p>
	<p>最后,提供一个配套的用户空间 <code>API</code> 代码,以帮助应用程序方便的访问零拷贝的编程接口。对应的收包和发包示例程序也包含在内。</p> <p>现在用户空间 <code>API</code> 的定义还是一个非标准的私有接口,未来可能会考虑发展成和 <code>libpcap</code> 接口一致。</p> <p><code>ntzc</code> 的原始代码可参照 <code>svn</code> 上的开源项目 http://code.google.com/p/ntzc/</p>

模块名称	网卡驱动	标识	ZC011
父模块	零拷贝模块	回溯标识	ZC01
模块功能	此模块为修改过的支持零拷贝的网卡驱动，数据包网卡接收到之后，并不会存放到系统默认的协议栈缓冲区，而是转移到 ntzc 的自定义缓冲区。这种功能的实现必然要依靠网卡驱动的支持。		
相关数据	网卡驱动的私有数据		
限制条件	此处的网卡驱动其实是本系统一个受限比较大的地方，由于零拷贝使用自定义的缓冲区，必然要依靠网卡驱动的支持，而由于内核版本的不同及硬件的不同，必然造成存在大量的网卡，因而如果更换了网卡或者升级了内核，必须同时改动网卡驱动。但幸运的是，本系统网卡驱动比较简单，若熟悉本系统，几个小时之内就可以改好一款可用驱动。		
输入数据	网卡数据包		
输出数据	填充好的 mbuf 结构体，每个 mbuf 对应一个数据包，此些数据包均存放在用户自定义缓冲区。		

<p>图例</p>	<div data-bbox="414 190 1085 660"></div> <p>以上为 mbuf 与数据包的对应关系，数据包存放在用户自定义的缓冲区，而 mbuf 即指向该数据包。</p>
<p>算法逻辑</p>	<p>本模块本质即是一个支持零拷贝的网卡驱动模块，数据包原先经过此网卡驱动时，使用很多类似 <code>skb_put</code>, <code>dev_alloc_skb</code> 等系统函数将数据包存放到内核默认缓冲区，而我们的零拷贝模块提供了另外一套自定义的缓冲区管理模块，同时提供了类如 <code>mbuf_put</code>, <code>dev_alloc_mbuf</code> 等函数，将原先的数据包存放到用户自定义的缓冲区。同时，填充 mbuf 的各个对象。</p>

调用接口	<p>调用了驱动接口模块的接口，主要接口列表如下：</p> <pre> unsigned char *mbuf_put(struct m_buf *mbuf, unsigned int len); unsigned char *mbuf_push(struct m_buf *mbuf, unsigned int len); unsigned char *mbuf_pull(struct m_buf *mbuf, unsigned int len); struct m_buf *nta_alloc_mbuf(struct net_device *dev, unsigned int length, gfp_t gfp_mask); void nta_kfree_mbuf(struct m_buf *mbuf); int nta_register_zc(struct net_device *netdev, int (*hard_start_xmit) (struct m_buf *mbuf, struct net_device *dev)); static inline void mbuf_reserve(struct m_buf *mbuf, int len); static inline unsigned int mbuf_headlen(const struct m_buf *mbuf); static inline unsigned char *mbuf_transport_header(const struct m_buf *mbuf); static inline void mbuf_set_transport_header(struct m_buf *mbuf, const int offset); static inline unsigned char *mbuf_network_header(const struct m_buf *mbuf); static inline void mbuf_set_network_header(struct m_buf *mbuf, const int offset); static inline unsigned char *mbuf_mac_header(const struct m_buf *mbuf); static inline int mbuf_transport_offset(const struct m_buf *mbuf); static inline u32 mbuf_network_header_len(const struct m_buf *mbuf); static inline int mbuf_network_offset(const struct m_buf *mbuf); static inline void mbuf_copy_to_linear_data(struct m_buf *mbuf, const void *from, const unsigned int len); static inline void mbuf_copy_from_linear_data_offset(const struct m_buf *mbuf, </pre>
------	--

提供接口	无
备注	<p>网卡驱动的修改算是系统中比较麻烦的一部分，因为网卡驱动的种类太多，所以每一个都要自己动手修改，当然，在实际使用中只要修改启用的网卡驱动即可。</p> <p>同样，尽管 <code>nta.h,nta.c</code> 中定义了大量的网卡驱动接口，在实际操作中仍然可能用到系统中未定义到的接口函数，此时需要用户自己来添加需要的接口函数，只需要将 <code>sk_buff</code> 的函数全部替换为 <code>mbuf</code> 的函数即可。</p> <p>开发人员在使用的过程中发现缺少更改驱动的接口,可按如下方案更改:</p> <p>a) 找到对应版本的内核源码中此函数的定义,推荐一个提供交叉引用搜索的网址: http://lxr.oss.org.cn/ident, 网络上很多与此类似的提供交叉引用,标识符查找的网站.</p> <p>b) 从中找到源定义,将 <code>sk_buff</code> 的全部操作替换为 <code>ntzc</code> 的 <code>m_buf</code> 操作,将此函数添加到 <code>nta.h</code> 中</p> <p>c) 将驱动中的 <code>sk_buff</code> 的操作替换为自定义的 <code>m_buf</code> 的操作。</p>

模块名称	驱动接口模块	标识	ZC021
父模块	零拷贝模块	回溯标识	ZC02
模块功能	此模块主要为网卡驱动提供必要的零拷贝接口，是连接网卡和自定义缓冲区的桥梁。网卡驱动通过使用此模块的接口，将网卡数据包转移到 ntzc 的缓冲区。		

<p>相关数据</p>	<p>最重要的依然是 <code>struct mbuf</code>,此结构体定义于 <code>nta.h</code>, 当接收到数据包时, 会填充此结构体, 提交到上一层使用, 因而此结构体及其重要, 但是此结构体已经在前面列出, 此处不再重复。</p> <p>其他重要的结构体如下:</p> <pre>struct mbuf_frag_struct { struct page *page; __u32 page_offset; __u32 size; }; union mbuf_shared_tx { struct { __u8 hardware:1, software:1, in_progress:1; }; __u8 flags; }; struct mbuf_shared_info { atomic_t dataref; unsigned short nr_frags; unsigned short gso_size; /* Warning: this field is not always filled in (UFO)! */ unsigned short gso_segs; unsigned short gso_type; __be32 ip6_frag_id; union mbuf_shared_tx tx_flags; unsigned int num_dma_maps; struct m_buf *frag_list; //struct mbuf_shared_hwtstamps hwtstamps; mbuf_frag_t frags[MAX_MBUF_FRAGS]; dma_addr_t dma_maps[MAX_MBUF_FRAGS + 1]; };</pre> <p>上述结构体均定义于 <code>nta.h</code>, 至于上述含义, 其实同样是把 <code>sk_buff</code> 的结构体替换为了 <code>m_buf</code> 的结构体, 本质上只是一份拷贝。</p>
-------------	--

限制条件	无
输入数据	本模块主要是联络网卡与内存管理模块的桥梁，主要起了一个“导流”的作用，将数据流从网卡中劫持到用户自定义的缓冲区。
输出数据	网络数据包
图例	无
算法逻辑	本模块的作用即是提供一系列接口给网卡驱动，在将数据包存入自定义缓冲区同时，填充 mbuf 结构体。
调用接口	<p>Bvl.h 中的一系列内存管理接口，比较重要的接口如下：</p> <pre> void *avl_alloc(unsigned int size, int cpu, gfp_t gfp_mask); void avl_free(void *ptr, int dir, int r_size); int avl_init(void); void avl_free_no_zc(void *ptr); int avl_init_zc(void); void avl_deinit_zc(void); void avl_fill_zc(struct zc_data *zc, void *ptr, int r_size); </pre>

提供接口	<p>主要接口列表如下：</p> <pre> unsigned char *mbuf_put(struct m_buf *mbuf, unsigned int len); unsigned char *mbuf_push(struct m_buf *mbuf, unsigned int len); unsigned char *mbuf_pull(struct m_buf *mbuf, unsigned int len); struct m_buf *nta_alloc_mbuf(struct net_device *dev, unsigned int length, gfp_t gfp_mask); void nta_kfree_mbuf(struct m_buf *mbuf); int nta_register_zc(struct net_device *netdev, int (*hard_start_xmit) (struct m_buf *mbuf, struct net_device *dev)); static inline void mbuf_reserve(struct m_buf *mbuf, int len); static inline unsigned int mbuf_headlen(const struct m_buf *mbuf); static inline unsigned char *mbuf_transport_header(const struct m_buf *mbuf); static inline void mbuf_set_transport_header(struct m_buf *mbuf, const int offset); static inline unsigned char *mbuf_network_header(const struct m_buf *mbuf); static inline void mbuf_set_network_header(struct m_buf *mbuf, const int offset); static inline unsigned char *mbuf_mac_header(const struct m_buf *mbuf); static inline int mbuf_transport_offset(const struct m_buf *mbuf); static inline u32 mbuf_network_header_len(const struct m_buf *mbuf); static inline int mbuf_network_offset(const struct m_buf *mbuf); static inline void mbuf_copy_to_linear_data(struct m_buf *mbuf, const void *from, const unsigned int len); static inline void mbuf_copy_from_linear_data_offset(const struct m_buf *mbuf, </pre>
------	---

备注	无
----	---

模块名称	内存管理模块	标识	ZC031
父模块	零拷贝模块	回溯标识	ZC03
模块功能	内存管理模块是整个零拷贝模块的核心模块，负责内核自定义缓冲区的分配，管理，释放；整个零拷贝模块即使围绕这一子模块完成数据包的整个流动。		

相关数据

```

struct avl_node
{
    unsigned long    value;
    struct avl_node_entry *entry;
};

struct avl_node_entry
{
    struct avl_node    **avl_node_array;
    struct list_head node_entry;
    u32    avl_entry_num;
    u16    avl_node_num, avl_node_pages;
    //u16    avl_node_pages;
};

struct avl_free_list
{
    struct avl_free_list    *next;
    unsigned int            size;
    unsigned int            cpu;
};

struct avl_chunk
{
    unsigned int            canary, size;
    atomic_t                refcnt;
};

struct avl_allocator_data
{
    struct avl_free_list    *avl_free_list_head;
    spinlock_t            avl_free_lock;
    struct list_head    avl_node_list;
    spinlock_t            avl_node_lock;
    u32    avl_node_entry_num;
    void    *zc_ring_zone;
};

```

限制条件	<p>此模块提供自定义的内存管理，用户可以自行调整缓冲区大小，当然，缓冲区越大，丢包率越低，这也就要求系统可用内存越大越好，对硬件提出了更高的要求。同时，此模块使用了固定大小的分区，环形缓冲区的每一块都是固定大小，必然存在一定的浪费。</p> <p>同时，调度策略的优劣直接影响系统的性能，系统的不完善必然造成无法充分发挥零拷贝的威力。</p>
输入数据	网卡数据包
输出数据	无
图例	<p>avl_allocator[cpu]</p> <p>cpu0 cpu1 cpu2</p> <p>Struct avl_allocator_data struct avl_free_list *avl_free_list_head; Struct list_head avl_node_list void *zc_ring_zone;</p> <p>Struct avl_node_entry struct avl_node **avl_node_array; Struct list_head Node_entry *</p> <p>Struct avl_node U_long value Struct avl_node_entry*entry *</p> <p>每个元素占1 page</p> <p>1<<BVL_ORD ER = 4</p> <p>Struct page Struct list_head {*next; *prev}</p> <p>Struct zc_control Struct zc_sniff sniffer[0] Struct zc_sniff sniffer[8] Struct net_device* netdev Struct net_device* netdev struct zc_data *zcb struct zc_ring_ctl *zc_ring struct timer_list test_timer *</p> <p>zc_ring_zone PAGE</p> <p>Zc_data PAGE</p> <p>struct zc_data data; _u32 Off; _u16 r_size; _u8 Entry; _u8 cpu:4; netdev_index:4;</p> <p>Struct avl_free_list Struct chunk Struct avl_free_list</p> <p>2048 2048 2048 2048</p> <p>Void* Data.ptr</p>

算法逻辑	<p>这是系统的核心模块，对照上图和源码，进行简单的讲解：</p> <p>模块加载之初，进行初始化，进行内存分配。每个 <code>cpu</code> 分配一个结构体 <code>avl_alloc_data</code>，其中建造了一个三维链表，以页为单位分配缓冲区。</p> <p>而 <code>struct zc_control</code> 将此缓冲区当作一个环形队列来处理，提供数据包缓冲区的分配，释放，提取等。这一部分属核心且复杂，但是对照上面的系统结构图，用户也可以很容易理解。</p>
调用接口	无
提供接口	<p>比较重要的一些接口如下：</p> <pre>void *avl_alloc(unsigned int size, int cpu, gfp_t gfp_mask); void avl_free(void *ptr, int dir, int r_size); int avl_init(void); void avl_free_no_zc(void *ptr); int avl_init_zc(void); void avl_deinit_zc(void); void avl_fill_zc(struct zc_data *zc, void *ptr, int r_size);</pre>
备注	<p>作为用户，无须理解此部分的构造；作为普通的开发者，若需要构建系统，同样可以不关心此部分；但若开发者系统对性能要求很高，不妨考虑改善此部分的内存分配测率，同样会对系统性能提高很多。</p>

模块名称	用户接口模块	标识	ZC041
父模块	零拷贝模块	回溯标识	ZC04
模块功能	<p>为内核缓冲区与用户的交互提供管理</p> <p>，将自定义缓冲区设置为虚拟设备(<code>/dev/zc</code>)缓冲区，用户可以通过映射（<code>MMAP</code>）此缓冲区来达到最网络数据包的访问。</p>		

相关数据

```
struct zc_data
{
    union {
        __u32    data[2];
        void      *ptr;
    } data;
    __u32        off;
    __u16        r_size; /* the packet size */
    __u8         entry;
    __u8         cpu:4, netdev_index:4;
};

#define ZC_MAX_ENTRY_NUM    170

struct zc_ring
{
    __u16 zc_pos;
    __u16 zc_used;
};

struct zc_ring_ctl{
    __u16 zc_used /* eraser */, zc_pos /* producer */;
    __u16  zc_prev_used, zc_dummy /* consumer */;
};

/*
 * Zero-copy allocation request.
 * @type - type of the message - ipv4/ipv6/...
 * @res_len - length of reserved area at the beginning.
 * @data - allocation control block.
 */
struct zc_alloc_ctl
{
    __u16    proto;
    __u16    res_len;
    struct zc_data  zc;
};
```

	<pre> struct zc_sniff { int sniff_id; int dev_index; int sniff_mode; #define ZC_SNIFF_NONE 0 #define ZC_SNIFF_RX 1 #define ZC_SNIFF_TX 2 #define ZC_SNIFF_ALL 3 u_int16_t pre_p; #define ZC_PRE_P_NPCP 0x8050 #define ZC_PRE_P_NORMAL 0 u_int16_t pre_type; #define ZC_PRE_P_ALL 0 #define ZC_PRE_P_CONRTOL 0x20 #define ZC_PRE_P_PACKET 0x10 #define ZC_PRE_P_SESSION 0x11 u_int32_t acl_index; }; </pre>
限制条件	<p>此部分模块将用户自定义缓冲区映射为虚拟混杂设备(/dev/zc)的设备缓冲区，此时，分配给的次设备号为 0，若 0 号被占用（目前测试过程中尚无测试到此冲突的出现，但是仍然存在可能），模块将加载失败。</p>
输入数据	网卡的数据包
输出数据	无（对外表现为设备缓冲区）

图例	无
算法逻辑	<p>此模块是将自定义缓冲区映射为虚拟混杂设备(miscdevice)的设备缓冲区，名称为“zc”</p> <pre>static struct miscdevice zc_gen_dev = { .minor = 0, .name = "zc", .fops = &zc_ops, };</pre> <p>同时定义了一系列的设备操作：</p> <pre>static struct file_operations zc_ops = { .poll = &zc_poll, .ioctl = &zc_ioctl, .open = &zc_open, .release = &zc_release, .read = &zc_read, .write = &zc_write, .mmap = &zc_mmap, .owner = THIS_MODULE, };</pre>
	通过对虚拟设备的一系列操作达到读写自定义缓冲区的目的,从而将数据包取出，供后续协议分析使用。
调用接口	无
提供接口	设备/dev/zc
备注	无

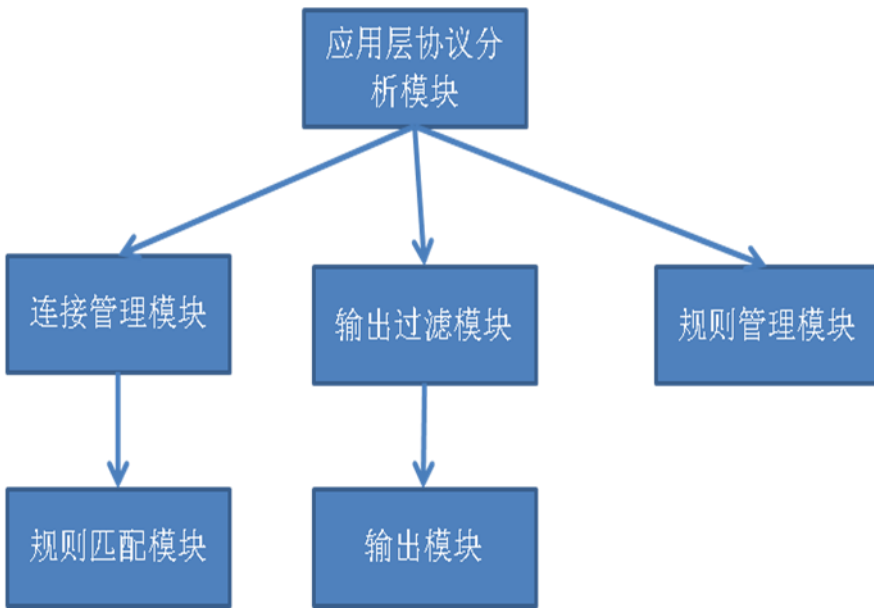
模块名称	用户接口	标识	ZC061
父模块	零拷贝模块	回溯标识	ZC06
模块功能	此模块名称为 用户接口，须注意此模块与“用户接口模块”不同，勿混淆。用户接口模块将内核自定义缓冲区设置为虚拟设备（/dev/zc）缓冲区，而用户接口则提供了一系列用户程序可直接使用的接口,在我们的系统中提供了三个类似 libpcap 的抓包接口，即存放在此模块中		
相关数据	zc_comm.h 中各个结构体，前面已述，此处不详解。		
限制条件	系统提供了若干接口函数，对于我们原先的系统，抓包使用了 libpcap 的各个函数，此处我们仿照 libpcap 各个函数，提供了三个主要的接口函数，但是没有提供完整的一套类 libpcap 函数，所以不得不采用 libpcap 与零拷贝共存的方案，并未能够完全舍弃 libpcap。这是系统的一个不足，但是，系统能够在零拷贝模式与 libpcap 轻量级抓包模式下切换，灵活性更强，也是系统的一个亮点。		
输入数据	无		
输出数据	抓取网卡的数据包		
图例	无		
算法逻辑	在初始化(zc_init)的过程中，映射（mmap）/dev/zc 的设备缓冲区（也是数据包缓冲区）到用户空间,然后通过用户接口函数对缓冲区操作，利用 zc_loop(一个类似 pcap_loop 的用户接口函数)循环将数据包提供给用户处理函数，系统处理完毕后利用 zc_destroy 将句柄销毁。		
调用接口	无		

提供接口	<p>/dev/zc 的设备操作接口，包括读，写，mmap 映射等</p> <pre>zc_t* zc_open_live(const char*dev, int snaplen, char* errbuf);</pre> <pre>void zc_destroy(zc_t* zc_ctl);</pre> <pre>int zc_loop(zc_t *zc_ctl, int cnt, void (*zc_handler)(unsigned char* user, const struct pcap_pkthdr *h, const unsigned char *bytes), unsigned char* user);</pre> <p>这三个函数的参数与 libpcap 的函数 pcap_open_live, pcap_destroy, pcap_loop 相同。</p>
备注	<p>系统尚未完善，仅提供了三个接口，所以必须与 libpcap 配合使用，若系统进行进一步开发，可以开发一整套 libpcap 的接口，到时，可完全舍弃 libpcap 这一轻量级的抓包工具，系统的稳定性也可以获得更大的提高。</p>

3.3.2 应用层协议分析模块

模块名称	应用层协议分析模块	标识	LA00
父模块	无	回溯标识	LA00
模块功能	该模块用于对一个数据包的应用层协议做分析，来发现当前网络中使用的常用软件。首先，系统读取由用户指定的待检测协议，将用于协议匹配的正则表达式加载入内存。然后，当数据包到达时，系统将数据包中应用层数据匹配正则表达式，如果匹配成功，则输出源 IP、源端口、匹配的时间、协议名到数据库		
相关数据	请见子模块		
限制条件	请见子模块		
输入数据	//以太网数据包 const char* pkt		

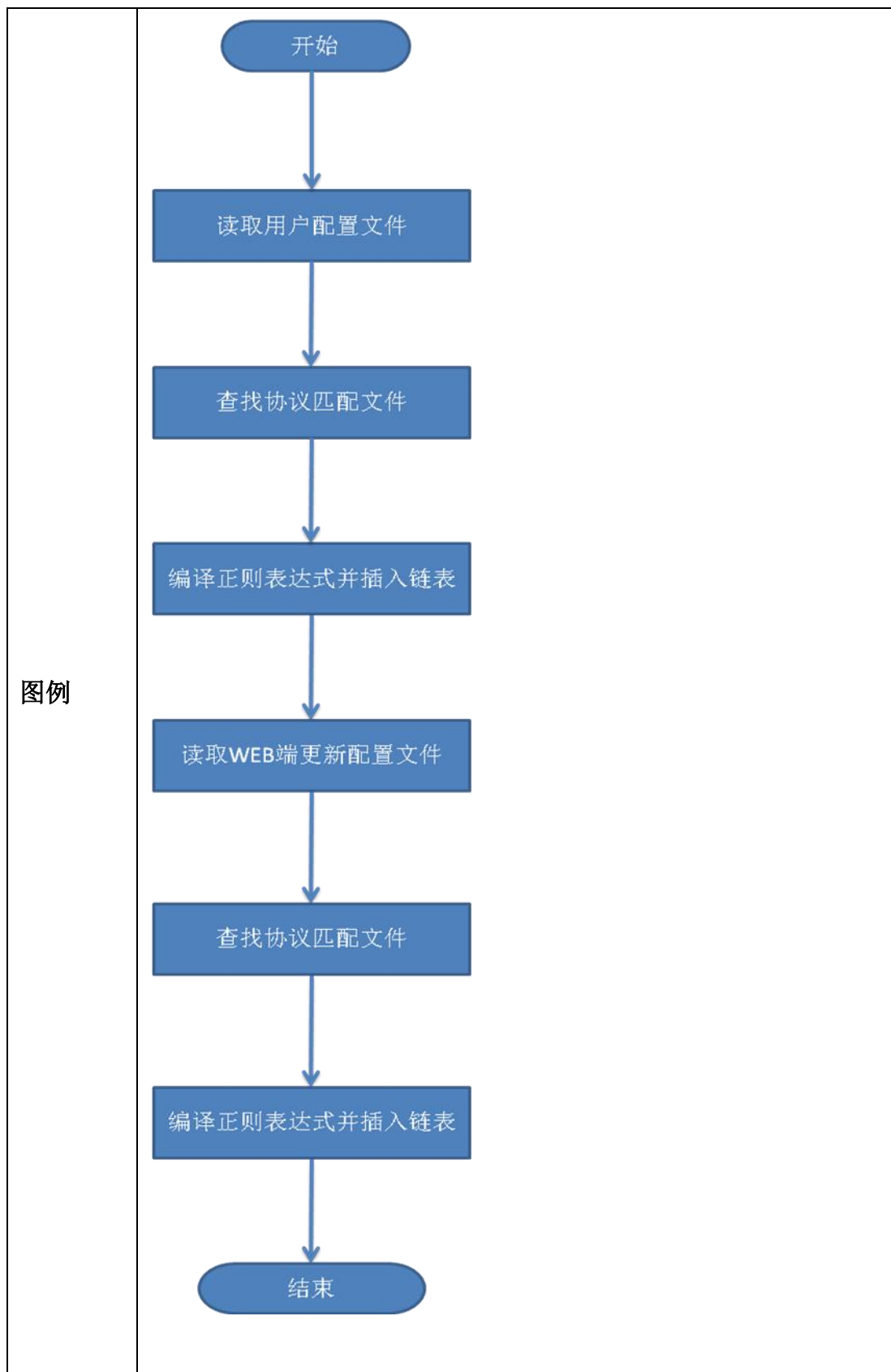
输出数据	<p>数据库表:</p> <pre> /*===== =====*/ /* Table: appproto */ /*===== =====*/ CREATE TABLE appproto (id INT UNSIGNED NOT NULL AUTO_INCREMENT, time INT UNSIGNED NOT NULL , ip INT UNSIGNED NOT NULL, port SMALLINT UNSIGNED NOT NULL, proto VARCHAR(255) NOT NULL, PRIMARY KEY(id)); id: 由 mysql 自动生成的记录标号，作为主键 time: 该次匹配成功的时间 ip: 局域网中用户所使用的 IP 地址 port: 局域网中用户使用该协议时所使用的端口号 proto: 局域网中用户使用的协议 </pre>
------	---

图例	<p>模块层次图</p>  <pre>graph TD; A[应用层协议分析模块] --> B[连接管理模块]; A --> C[输出过滤模块]; A --> D[规则管理模块]; B --> E[规则匹配模块]; C --> F[输出模块];</pre>
算法逻辑	<pre>///伪代码 ParsePatternFiles(); //规则管理模块 if (ParsePacket(const char* pkt)) //连接管理、规则匹配模块 { if (FiltrateOutput()) //输出过滤模块 { OutPut(); //输出模块 } }</pre>

调用接口	<pre> /// 连接管理模块 /// 提供连接管理和规则匹配功能 /// @param pkt 以太网数据包，从以太帧头部开始 /// @param srcip 分析后返回的源 IP 地址 /// @param srcport 分析后返回的源端口 /// @param proto 分析后返回的协议名 /// @return 协议号 int ParsePacket(const char* pkt, int* srcip, short* srcport, char** proto); /// 输出过滤模块 /// 提供匹配结果的过滤功能 /// @param srcip 源 ip 地址 /// @param proto 协议名 /// @return 返回 0，数据不需要输出；返回 1，数据需要输出 int FiltrateOutput (unsigned int srcip, char* proto); /// 规则管理模块 /// 提供规则的管理 /// @param filename 配置文件的文件名 void ParseConfigurationFile(const char* fileName); </pre>
提供接口	<pre> /// 应用层协议分析模块 /// 提供对数据包的应用层进行协议分析的功能 /// @param pkt 以太网数据包，从以太帧头部开始 void AppprotoDetection(cosnt char* pkt); </pre>
备注	无

模块名称	规则文件管理模块	标识	LA011
父模块	应用层协议分析模块	回溯标识	LA01
模块功能	1. 读取本地规则配置文件。 2. 根据配置文件查找指定协议的规则文件。 3. 将规则文件中的正则表达式编译并加载入内存 4. 读取 WEB 端规则配置文件 5. 根据配置文件查找指定协议的规则文件。 6. 将规则文件中的正则表达式编译并加载入内存		
相关数据	<pre> /// 该结构定义了规则的存储 struct Pattern { int mark; ///< 表示该规则对应的协 议号 int eflags; ///< 处理正则表达式用到 的标识 int cflags; ///< 处理正则表达式用到 的标识 char* name; ///< 协议名 char* pattern_string; ///< 编译后的正则表达式 regex_t preg; ///< 处理正则表达式用到 的句柄 }; </pre>		

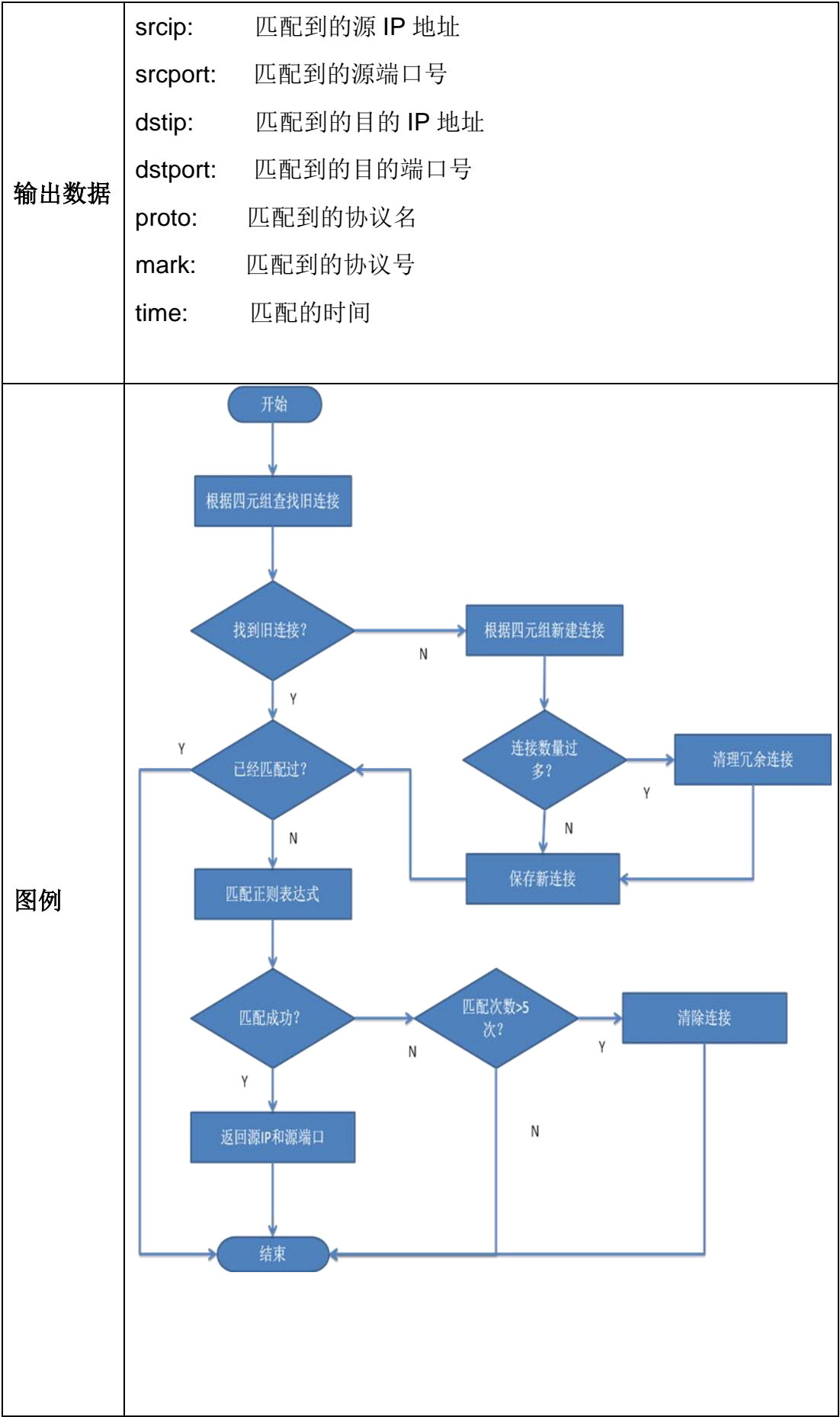
限制条件	<p>该模块用到了正则表达式库，对读入的正则表达式进行编译，为后面正则表达式的匹配打下基础。</p> <p>本地规则配置文件：由管理员在本地指定的需要检测的应用层协议名</p> <p>WEB 端规则配置文件：由管理员在 WEB 端指定的需要检测的应用层协议名</p> <p>本地规则文件夹：由管理员在本地指定的需要检测的规则文件集</p> <p>WEB 端规则文件夹：由管理员在 WEB 端指定的需要检测的规则文件集，其通过数据库来获取</p>
输入数据	//配置文件名 const char* filename
输出数据	Pattern 结构链表



算法逻辑	<pre> /// 伪代码 while (从配置文件中获取一行) { 分析当前行 if (当前行为空格或者注释) { continue; } else { 找到并分析规则文件; 获取正则表达式并挂到链表上 } } </pre>
调用接口	无
提供接口	<pre> /// 规则管理模块 /// 提供规则的管理 /// @param filename 配置文件的文件名 void ParseConfigurationFile(const char* fileName); </pre>
备注	无

模块名称	连接管理模块	标识	LA021
父模块	应用层协议分析模块	回溯标识	LA02

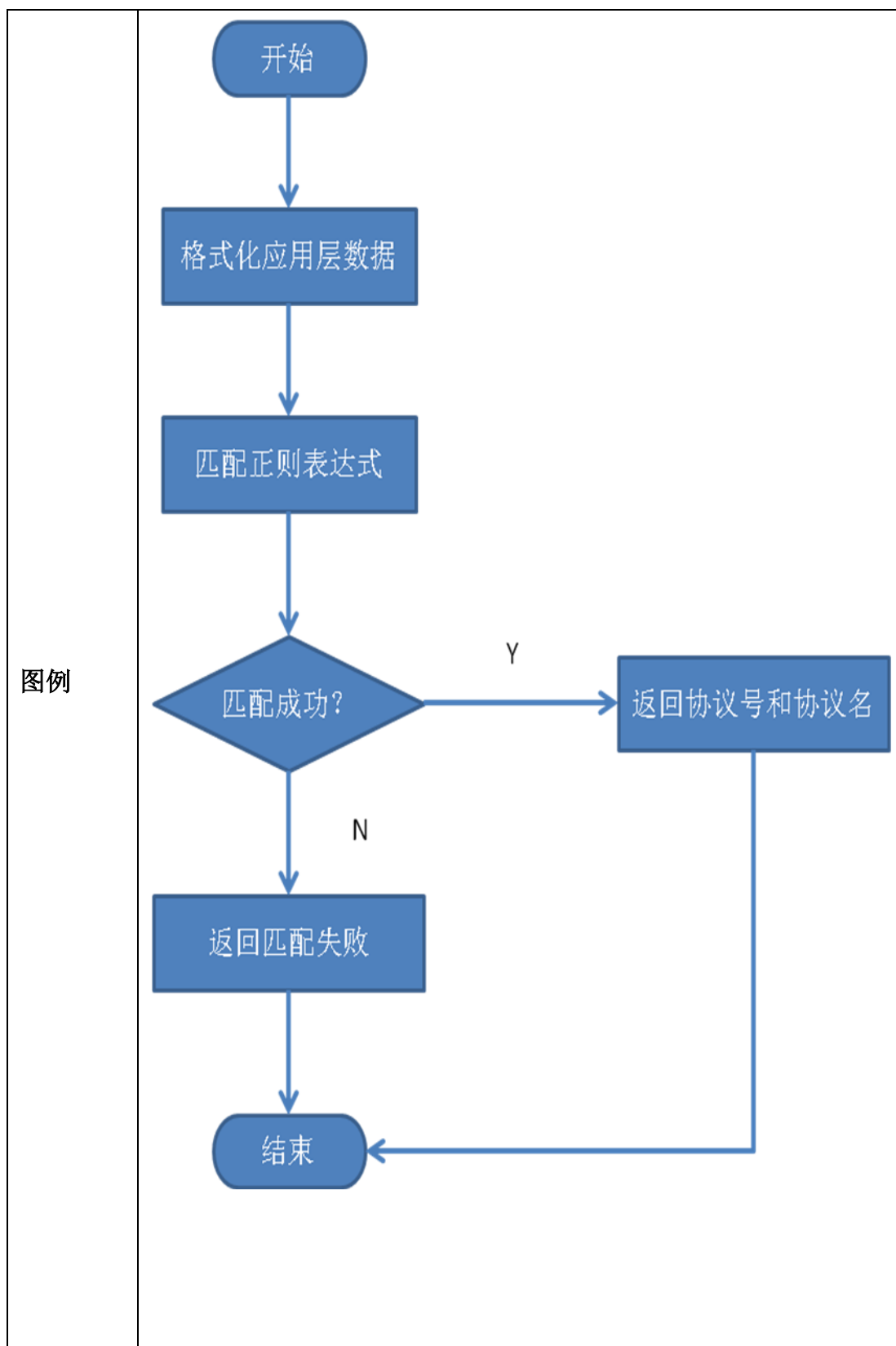
模块功能	<p>为了提高正则表达式匹配速度的效率，引入连接管理的概念。首先把一个数据包的源 IP 地址、源端口号、目的 IP 地址、目的端口看成一个整体，这个四元组唯一标识了一个连接，当一个连接连续匹配失败次数超过 5 次，则清除匹配失败的连接，使连接信息不会占用过多内存，为后面正则表达式匹配打下基础。</p> <p>过程：</p> <ol style="list-style-type: none"> 1. 查找当前连接（由源 IP,源端口号,目的 IP,目的端口号组成）是否是旧连接，若是，则读取旧连接，否则新建一个连接。 2. 查看该连接是否已经匹配成功,若匹配成功,则跳过后续操作 3. 若该连接上数据未匹配成功过,则遍历正则表达式链表，匹配当前连接上应用层的数据 4. 若匹配成功，则输出源 IP、源端口号、协议名、匹配成功的时间 5. 若连续匹配失败超过 5 次，则删除该连接
相关数据	<pre> /// 使用的底层数据结构 map<char[12], struct Connection*> /// 该结构定义了一个连接的表示和存储 struct Connection { unsigned int num_packets; ///< 该连接上连续匹配包的数量 unsigned int mark; ///< 该连接上匹配的协议号 char key[12]; ///< 该连接的索引号 }; </pre>
限制条件	Snort: snort 提供的以太网数据包作为输入
输入数据	const char* pkt;



算法逻辑	<pre>///伪代码 初始化 map 结构; if (数据包是 tcp 或者 udp) { map 查找当前连接; if (当前连接是旧连接) { 继续操作; } else { 建立新连接; } if (当前连接已经匹配成功) { return; } else if (当前连接匹配次数 <= 5) { 匹配正则表达式; if (匹配到的协议号 > 2) { 返回源 IP 和源端口; } } else { 从 map 中删除当前连接; } }</pre> <div>第 80 页</div>
------	---

调用接口	<pre> /// 规则匹配模块 /// @param data 待匹配的应用层数据 /// @param datalen data 的长度 /// @param proto 匹配到的协议名 /// @return 匹配到的协议号 int Classify(char* data, int datalen, char** proto); </pre>
提供接口	<pre> /// 连接管理模块 /// 提供连接管理和规则匹配功能 /// @param pkt 以太网数据包，从以太帧头部开始 /// @param srcip 分析后返回的源 IP 地址 /// @param srcport 分析后返回的源端口 /// @param proto 分析后返回的协议名 /// @return 协议号 int ParsePacket(const char* pkt, int* srcip, short* srcport, char** proto); </pre>
备注	无

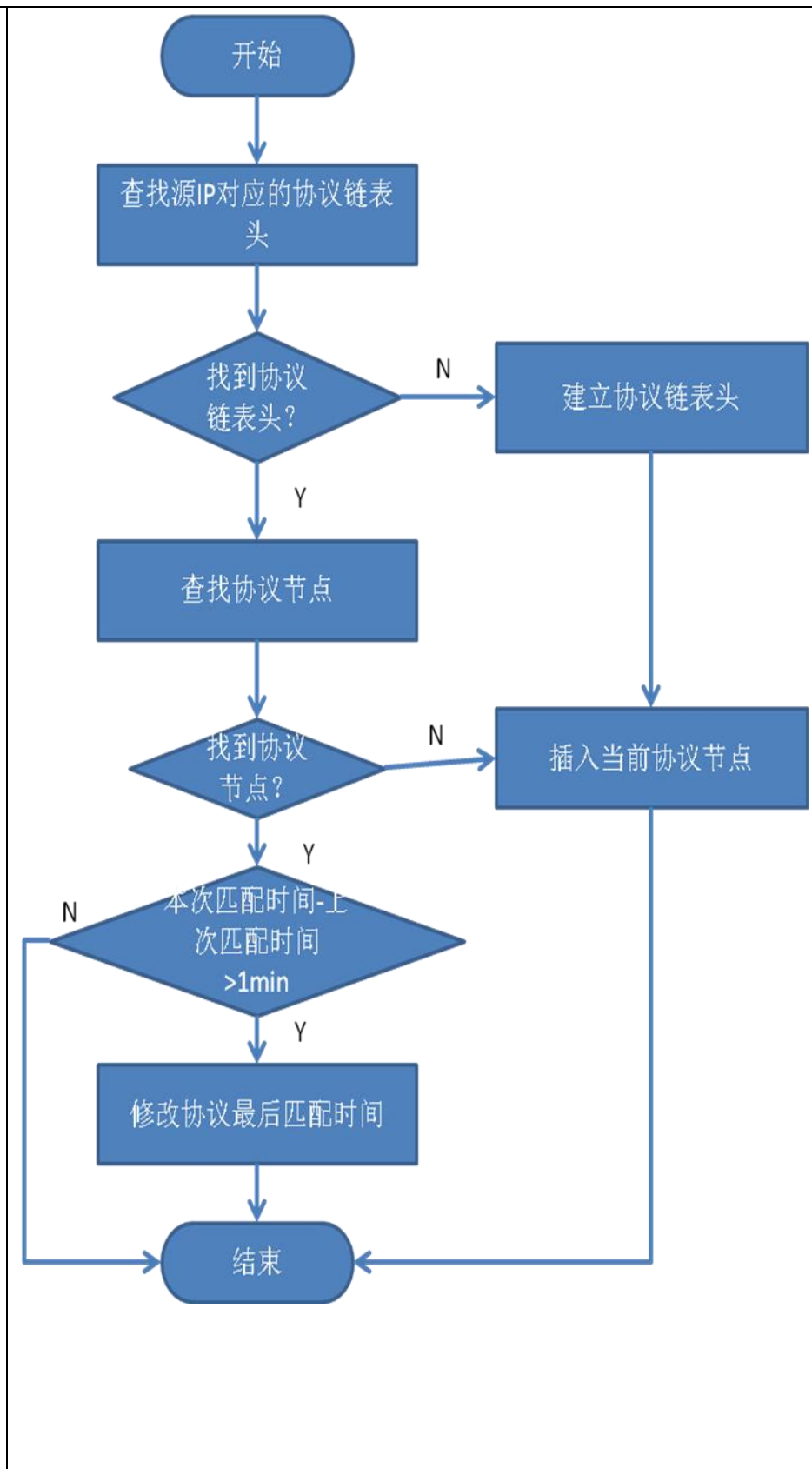
模块名称	规则匹配模块	标识	LA031
父模块	连接管理模块	回溯标识	LA03
模块功能	1. 去除应用层数据中的 0x00，防止匹配不完整 2. 遍历正则表达式链表，逐个匹配正则表达式 3. 若匹配成功，则返回协议号和协议名称 4. 若匹配不成功，返回协议号为 0		
相关数据	无		
限制条件	无		
输入数据	char* data; //待匹配的应用层数据 int datalen,; // data 的长度 char** proto; //匹配到的协议名		
输出数据	匹配到的协议号和协议名		



算法逻辑	<pre> /// 伪代码 去除 data 中的 0x00; while (规则链表中当前项合法) { if (data 匹配正则表达式得到的协议号 > 2) { 保存协议名; break; } 检查下一项; } return 协议号; </pre>
调用接口	无
提供接口	<pre> /// 规则匹配模块 /// @param data 待匹配的应用层数据 /// @param datalen data 的长度 /// @param proto 匹配到的协议名 /// @return 匹配到的协议号 int Classify(char* data, int datalen, char** proto); </pre>
备注	无

模块名称	输出过滤模块	标识	LA041
父模块	应用层协议分析模块	回溯标识	LA04
模块功能	<p>为了提高管理员查看数据的效率，匹配结果需要被过滤后才插入数据库。过滤算法使用 map 作为底层数据结构，局域网中用户 IP 作为 key，将该用户所使用的协议信息存为链表作为 value。当某个用户在某个时间段内连续使用某一软件，则数据只被记录一次，这样很大减少数据库的负担，也使得数据库数据清晰明了，降低管理员的负担</p> <p>过程：</p> <ol style="list-style-type: none"> 1.检查当前检测到的协议是否已经检查过 2.若检查过，并且超过了检查间隙时间，则更新检查时间；反之，跳过后续操作 3.若没检查过，则将记录下当前协议名和匹配时间等数据 		
相关数据	<pre> /// 使用的底层数据结构 map<unsigned int ip, struct Info*>; /// 该结构定义了一个 ip 上所使用的应用层协议信息 struct Info { time_t time; ///< 最近使用当前协议的时间 char* proto; ///< 协议名 struct Info* next; ///< 指向下一个 info }; </pre>		
限制条件	无		
输入数据	<pre> unsigned int srcip; //匹配到的源 IP char* proto; //匹配到的协议名 </pre>		
输出数据	过滤的结果（源 IP，源端口、协议名、匹配时间）		

图例



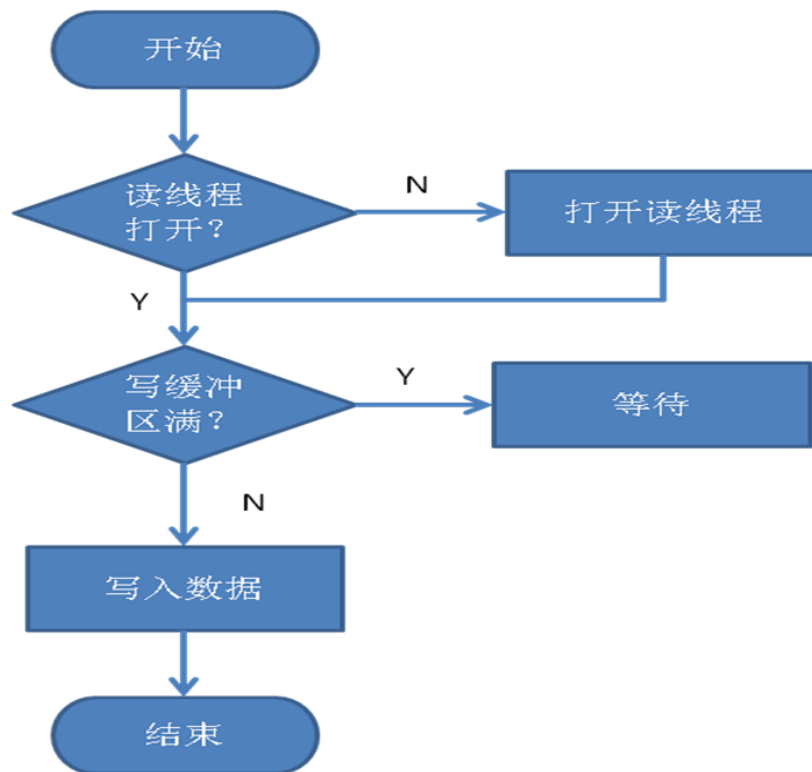
<p>算法逻辑</p>	<pre> 查找源 IP 对应的协议链表头; if (找到协议链表头) { 查找协议节点; if (找到协议节点) { if (本次匹配时间 - 上次匹配时间 > 1 分钟) { 修改协议最后匹配时间; return 1; } else { return 0; } } else { 插入当前节点; return 1; } } else { 建立协议链表头; 插入当前节点; return 1; } </pre>
-------------	---

调用接口	无
提供接口	<pre> /// 输出过滤模块 /// 提供匹配结果的过滤功能 /// @param srcip 源 ip 地址 /// @param proto 协议名 /// @return 返回 0，数据不需要输出；返回 1，数据需要输出 int FiltrateOutput (unsigned int srcip, char* proto); </pre>
备注	无

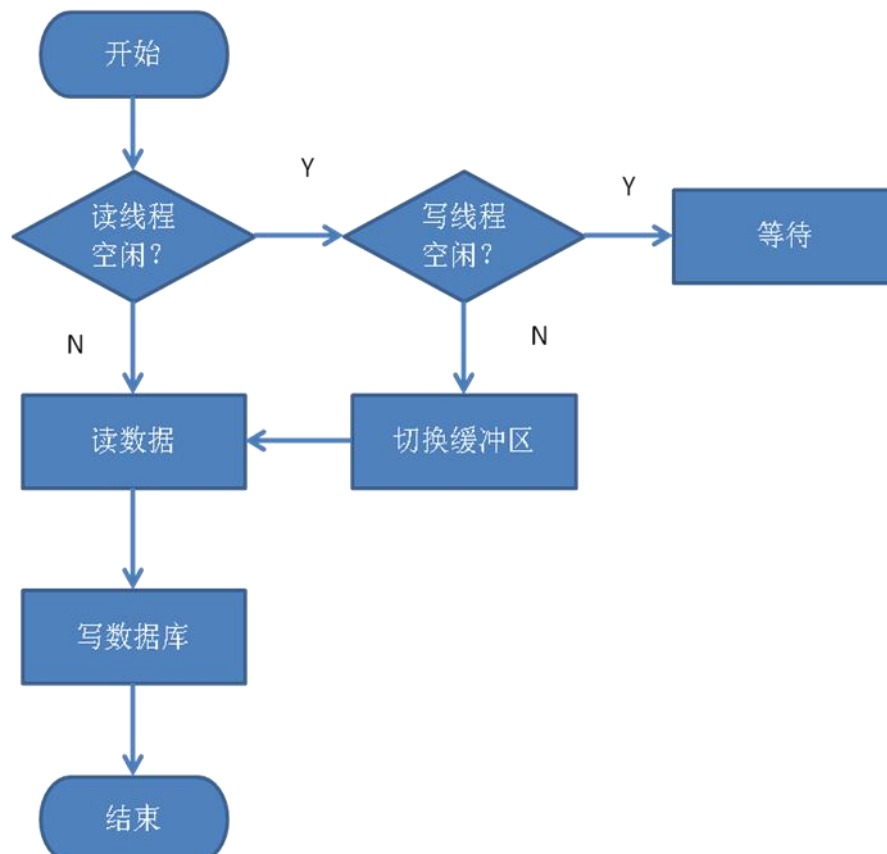
模块名称	输出模块	标识	LA042
父模块	应用层协议分析模块	回溯标识	LA04
模块功能	<p>为了提高数据库插入操作的效率，提高数据匹配和数据库 IO 操作的并行性，数据必须进行缓冲。双缓冲使用 2 个固定大小的数组作为 2 条循环队列，由 2 个应用级线程来管理，采用读线程优先算法，使得当读线程空闲时就进行队列切换，无需等待另外队列装满，比起单缓冲，双缓冲极大减少了读写的同步开销，提高了匹配和输出的并行性。</p> <p>过程：</p> <p>写线程将数据拷贝到由写指针指定的缓冲区</p> <p>读线程从由读指针指定的缓冲区读取数据，并执行数据库 IO 操作。</p>		
相关数据	<pre> /// 该结构表示缓冲区的元素 struct Insertion { unsigned int time; ///< 匹配时间 unsigned int ip; ///< 源 IP unsigned short port; ///< 源端口 char proto[255]; ///< 协议名 }; </pre>		
限制条件	Mysql: 分析出的结果通过 mysql 输出		
输入数据	<pre> //数据库输出需要的结构 struct Insertion _insertion </pre>		
输出数据	数据库操作语句字符串		

图例

写线程:



读线程:

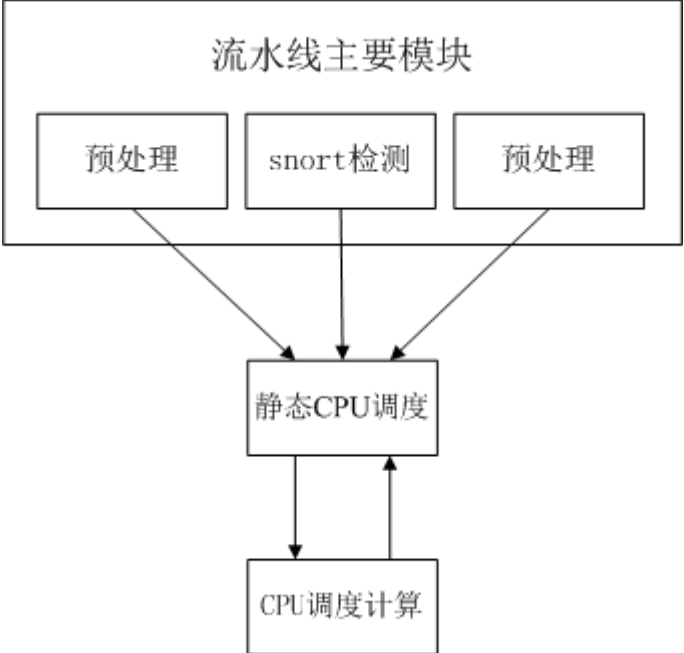


算法逻辑	<pre>/// 伪代码(写线程) if (读线程未打开) { 打开读线程; } if (写缓冲区满) { sleep(1); } else { 写入当前数据; } /// 伪代码(读线程) if (读线程空闲) { if (写线程空闲) { sleep(1); } else { 加锁; 切换队列; 解锁; } } else { 读取数据;</pre>
------	---

调用接口	无
提供接口	<pre>/// 提供存放缓冲区的功能 /// @param _insertion 数据库输出需要的结构 Q_PUT(struct Insertion _insertion)</pre>
备注	无

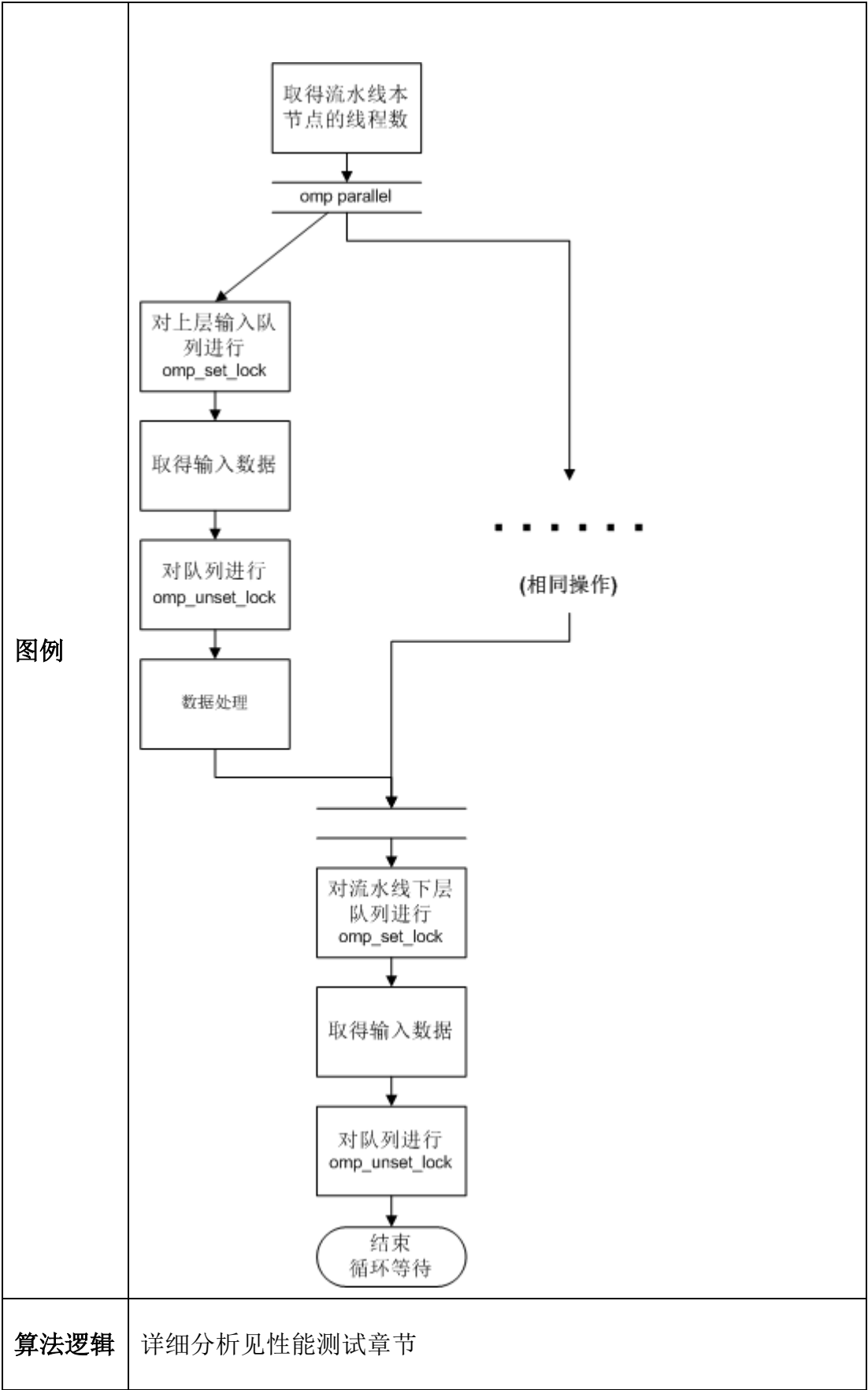
3.3.3 多核调度模块

模块名称	静态 CPU 调度模块	标识	MC001
父模块	无	回溯标识	MC00
模块功能	根据不同的 CPU 数目，利用已定义的静态均衡算法，对进程进行 CPU 绑定，以优化程序性能。		
相关数据	kmp_affinity_mask_t —— 进行亲和力设置的变量类型		
限制条件	要求 Intel C++ Compiler 提供支持，利用 Openmp 接口进行亲和力设置。静态均衡算法在过多（如超过 16）的 CPU 情况下，不能让程序达到最优		
输入数据	用整型表示流水线中对应的节点代号，如预处理、入侵检测等		
输出数据	CPU 静态调度方案		

<p>图例</p>	<p>与其他模块间的关系：</p> <div data-bbox="391 324 1077 974"><p>流水线主要模块</p><div><div>预处理</div><div>snort检测</div><div>预处理</div></div><div>静态CPU调度</div><div>CPU调度计算</div></div>
<p>算法逻辑</p>	
<p>调用接口</p>	<p>以下由 openmp 提供接口</p> <ol style="list-style-type: none">1.omp_get_num_procs ()，取得 CPU 数量2. kmp_create_affinity_mask() ， 创建 初始化 一个 kmp_affinity_mask_t3. kmp_set_affinity_mask_proc() ， 把对应的 CPU 加入 kmp_affinity_mask_t <p>以下由 CPU 调度计算子模块提供接口</p> <ol style="list-style-type: none">1. as2_get_affinity_cpus()，取得 CPU 均衡方案

提供接口	<pre>void as2_set_affinity(int node)</pre> <p>作用：自动设置 node 所对应的亲和力，以达到 CPU 绑定效果</p> <p>参数：node，</p> <p>0 代表预处理模块，1 代表 snort 检测模块，2 代表应用层协议分析模块</p>
备注	<p>在对应模块主线程的初始化时，调用一次上述接口</p>

模块名称	CPU 调度计算模块	标识	MC002
父模块	静态 CPU 调度模块	回溯标识	MC00
模块功能	根据不同的 CPU 数目，利用已定义的静态均衡算法，返回其调度方案		
相关数据	无		
限制条件	无		
输入数据	<p>分别是 node(int)，start(int *)，end(int *)，</p> <p>其中 node 表得对应的模块，start 表示分配的起始 CPU，end 表示分配的最后 CPU，start>=end。</p>		
输出数据	模块将修改 start、end，以达到返回其调度方案		



调用接口	<p>以下由 openmp 提供接口</p> <p>1.omp_get_num_procs (), 取得 CPU 数量</p>
提供接口	<p>void as2_get_affinity_cpus(int node,int* start,int* end)</p> <p>作用：利用 node 标识，取得对应的调度方案，以 start 和 end 返回</p> <p>参数：node，见父模块，start 表示分配的起始，end 表示分配的最后</p>
备注	<p>由 静态 CPU 调度模块 调用，不必主动调用。</p>

3.3.4 WEB 模块

(1) 整体框架流程:

登录模块->控制操作模块

登录模块:

用户进行登录

控制操作模块:

用户管理模块: 查看、删除、修改用户密码, 添加用户

权限: 管理员账户

规则管理模块: 查看、删除、添加规则

权限: 管理员账户

日志管理模块:

(1) 总体日志信息: sensor 数, Alert 数等

(2) 入侵图形分析: 如, 时间-Alerts 数等

(3) 应用层协议分析: 迅雷的协议, QQ 等

权限: 管理员账户和普通用户

(2) 数据库

详见 3.1.6 数据库设计

(3) 具体类与页面设计

1 具体函数类

主要有以下几个类，**ag_session**(会话类)，**page**（页面管理类），**user**（用户管理类），**alerts**（警告类），**utility**（工具类），**graph_data**（图形数据类），**aproto**(用户层协议类)，**rule**(规则类)，**system_info**(系统信息类)

2 页面文件设计

index.php

:默认主页面，登录界面。

登录成功将根据不同用户跳转到，**admin.php** 页面（管理员账户页面）或者 **user.php** 页面（一般用户界面）

登录失败，如用户名密码错误将在此页面上显示登录错误。

admin.php

管理员主操作页面。

管理员账户登录成功后，会跳转到该页面。

user.php

普通用户操作页面。

普通用户登录成功后，会跳转到该页面

includes 目录下：

config.php

配置文件，初始化后面需要的实体类（**ag_session**, **user**, **alert** 等类），及进行数据库的连接，保存会话。

config.php 由 **index.php** 登录页面调用。

connection_setting.php

数据库连接配置文件，设置数据库账户，密码，数据库名称等。

session_settings.php

会话变量的配置，会话时间及会话回收时间。

jpggraph_setting.php

图像显示大小的配置。

classes 目录下

该目录下为实体类的文件，以及其他依赖库文件，如 adodb, jpggraph

session_class.php

class ag_session（会话类）

作用：管理系统会话，控制用户登录，注销已经数据库访问的链接

变量：

\$ag_session_id 会话 id

\$session_time 会话时间

\$session_gc_time 会话垃圾回收时间

\$DB 数据访问的链接

重要函数：

ag_session(\$session_time,\$session_gc_time,\$DB)

:构造函数

auth_get_status()

:如果可能（如存在 cookie）自动登录

login(\$username,\$password)

:登录

set_log(\$user_id,\$type_auth)

:记录登录日志

logout()

:注销

page_class.php

class page (页面类)

作用：html 语言页面的显示，方便管理页面头部和底部

函数：

page()

:构造函数

getHeader(\$interface,\$title)

:返回 html 头部代码，\$title 为 title 名称，\$interface 为导入的 style 文件

getFooter()

:返回</body></html>

get_credits()

:返回版权信息

utility_class.php

class utility (工具类)

作用：提供一些常用的功能函数

函数：

utility()

:构造函数

get_logo(\$link)

:返回 logo 图片的链接

user_class.php

class user (用户类)

作用：管理用户,提供添加用户，修改用户信息，删除用户操作

变量：

var \$DB;

重要函数：

user(\$DB)

:构造函数

exist_user(\$id)

:判定某个用户是否存在
check_user(\$user)
:判定用户名是否有效
check_password(\$password)
:判定密码是否有效
check_email(\$email)
: 判断邮箱名是否有效
id_to_user(\$id)
:根据用户 id 返回用户名
user_to_id(\$user)
:根据用户名返回用户 id
add_user(\$user_array)
:添加用户
change_password(\$id,\$password)
:修改用户密码
del_user(\$id)
:删除用户
set_user(\$user_array)
:修改用户的信息

rule_class.php

class rule (规则类)
作用:管理规则，提供规则的添加，修改及删除
变量:
\$DB
函数:
rule(\$DB)
:构造函数
exist_rule(\$id)

:判断该规则是否存在

`add_rule($rule_array)`

:添加规则

`change_rule($id,$rule)`

:修改规则

`del_rule($id)`

:删除规则

`get_rule($id)`

:返回该条规则

`get_rules()`

:返回显示数据库存在的规则

`option_list_admin($id)`

:admin 页面中的规则操作列表

`option_list_user($id)`

:user 页面的规则操作列表

alerts_class.php

`class alerts(警报类)`

作用：提供显示多种警报形式，如总警报数，某个 ip 的警报数，某个端口的警报数

变量：

`$DB;`

函数：

`alerts($DB)`

:构造函数

`sensorCnt()`

:传感器数

`eventCnt()`

:总警报数

`eventBySensor($sensorID)`

:某个传感器上的警报数

uniqueAlertCnt()

:总的警报类型数

uniqueAlertCntBySensor(\$sensorID)

:某个传感器上的总的警报类型数

uniqueIpDstCnt()

:产生警报的目的 ip 地址数目

uniqueIpSrcCnt()

:产生警报的源 ip 地址数目

uniqueLinkCnt()

:产生警报的链接数

uniqueSrcPortCnt()

:产生警报的源端口数目

uniqueDstPortCnt()

:产生警报的目的端口数目

uniqueTcpSrcPortCnt()

:产生警报的 Tcp 源端口数目

uniqueTcpDstPortCnt()

:产生警报的 Tcp 目的端口数

uniqueUdpSrcPortCnt()

:产生警报的 Udp 源端口数目

uniqueUdpDstPortCnt()

: 产生警报的 Udp 目的端口数

UDPPktCnt()

:与 Tcp 相关的报警数

UDPPktCnt()

:与 Udp 相关的报警数

PortscanPktCnt()

:与端口扫描相关的报警数

graph_data_class.php

```
class graph_data
```

作用：用于产生 general_display.php 文件需要的显示数据

变量：

`$DB;`

函数：

```
graph_data($DB)
```

 :构造函数

```
getTimeDataSet(&$xdata, $chart_type, $time_start, $time_end)
```

 :返回时间-警报数的二维数组的个数

`$xdata` 保存改二维数组

`$chart_type` 图表类型

`$time_start` 时间的开始

`$time_end` 时间段的结束

```
getIpDataSet(&$xdata, $chart_type, $time_start, $time_end)
```

 : 返回 ip-警报数的二维数组的个数

`$xdata` 保存改二维数组

`$chart_type` 图表类型

`$time_start` 时间的开始

`$time_end` 时间段的结束

```
getPortDataSet(&$xdata, $chart_type, $time_start, $time_end)
```

 :返回 port-警报数的二维数组的个数

`$xdata` 保存改二维数组

`$chart_type` 图表类型

`$time_start` 时间的开始

`$time_end` 时间段的结束

```
getSensorDataSet(&$xdata, $chart_type, $time_start, $time_end)
```

: 返回 **sensor**-警报数的二维数组的个数

\$xdata 保存改二维数组

\$chart_type 图表类型

\$time_start 时间的开始

\$time_end 时间段的结束

agLong2IP(\$long_IP)

:将长整形的 **ip** 地址转换成点分制的 **ip** 地址

getSignatureName(\$sig_id, \$db)

:返回警报类型的名称

checkTime(\$time_group)

:检查时间是否为空，及时间格式是否正确

StoreAlertNum(\$sql, \$label, &\$xdata, &\$cnt, \$min_threshold)

:将时间-报警数存为二维数组形式

\$sql 执行的 **sql** 语句

\$label x 轴坐标显示的标签

\$xdata 返回的二维数组

\$cnt 二维数组的个数

\$min_threshold x 轴的最小间距

approto_class.php

class approto

作用：显示用户层协议相关信息

变量

\$DB

函数

getProtoDataSet(&\$xdata)

:数目-Proto 二维数组形式

:\$xdata 返回二维数组

返回值为二维数组的个数

system_info_class.php

class system_info

作用：显示系统的信息

函数

system_info()

:构造函数

GetkernelVersion()

:系统的内核版本

GetUptime()

:更新时间

interfaces 目录下

admin 目录下

管理员操作相关的文件

manage_user.php

add_user.php

manage_user.php 页面提供管理用户，查看、删除、修改用户信息，调用 add_user.php 页面进行用户的添加。

manage_rules.php

add_rule.php

manage_rules.php 页面管理规则，查看、删除、修个规则，调用 add_rule.php 页面进行规则的添加。

info_system.php

显示相关的系统信息

common 目录下

general_info.php

显示主页的报警信息

general_display.php

主页图表显示 tcp,udp,icmp,portscan 的报警信息

view_logs.php 日志查看文件与下面 5 个相关

graph_main.php

graph_form.php

graph_display_bar.php

graph_display_line.php

graph_display_pie.php

日志查看中的图表显示

graph_main.php 主的逻辑控制页面，控制图形的显示，如果数据库中没有数据，将不进行图形显示，并报告没有数据。

如果数据库中存在可疑显示的数据，将由 **graph_form.php** 传递的参数然后发送参数给 **graph_display_*.php** 不同文件来显示不同的图形

view_proto.php 应用层协议信息参考相关的文件

proto_display.php

控制图形的显示，如果没有数据，将不进行图形显示。

proto_disply_pie.php 饼图

目前 **view_proto** 只能显示不同协议所占得比例，用饼图来表示。

lang 目录下

ch-dictionary 中文语言

en-dictionary 英文语言

第四章 性能测试

4.1 测试方案

AegisShield 基于多核处理平台的网络行为分析系统有几大模块组成：零拷贝模块，snort 核心，网络行为分析模块，多核处理模块，web 部分。我们将对各大模块逐一进行模块测试，后测试整体的性能。

4.2 测试环境

下面列出的为我们的系统整体的运行环境，对于个别测试，如零拷贝模块的性能测试，我们采用了不同的测试配置，对于此类测试，在测试案例中我们会单独予以说明，环境无特别说明的为在系统运行环境中进行测试。

硬件环境：

处理器	Intel(R) Core(TM)2 CPU T5450 @ 1.66GHz (2 CPUs),
内存	2048MB RAM DDR-III 800
网络适配器	Marvell Technology Group Ltd. 88E8039 PCI-E Fast Ethernet Controller Intel Corporation PRO/Wireless 3945ABG [Golan] Network Connection

软件环境：

操作系统	CentOS 5.4 (core 2.6.32)
编译器	GCC 4.4.3
Web 服务器	Apache 2.2
PHP 模块	PHP 5
数据库服务器	mysql Ver 14.14 Distrib 5.1.41

4.3 环境搭建

安装之前，请把 CentOS 自带的 php, mysql, apache 删除

```
Yum remove php,mysql,httpd
```

,并且准备好以下要安装程序的源码包。

安装 Zlib,libpcap,libxml2,libpng,gd,jpeg

```
./configure
```

```
Make
```

```
Make install
```

安装 mysql

```
# groupadd mysql
```

```
# useradd -g mysql mysql
```

```
# tar -zxvf mysql
```

```
# cd mysql
```

```
# ./configure --prefix=/usr/local/mysql --with-charset=gb2312
```

```
# make
```

```
# make install
```

```
# cp support-files/my-medium.cnf /etc/my.cnf
```

```
# cd /usr/local/mysql
```

```
# bin/mysql_install_db --user=mysql
```

```
# chown -R root .
```

```
# chown -R mysql var
```

```
# chgrp -R mysql .
```

```
# bin/mysqld_safe --user=mysql &
```

```
#gedit /etc/ld.so.conf
```

在文件最后加入 2 行:

```
/usr/local/mysql/lib/mysql
```

```
/usr/local/lib
```

```
# ldconfig
```

安装 **DBD-mysql**

```
# cd DBD-mysql
# export LANG=C
# perl Makefile.PL \
# --libs="-L/usr/local/mysql/lib/mysql -lmysqlclient -lz" \
# --cflags=-I/usr/local/mysql/include/mysql \
# --testhost=127.0.0.1 \
# --mysql_config=/usr/local/mysql/bin/mysql_config
# make
# make install
```

设置 **mysql** 自启动

```
# cp /usr/local/mysql/share/mysql/mysql.server /etc/init.d/mysql
# chmod 755 /etc/init.d/mysql
# cd /etc/rc3.d
# ln -s /etc/init.d/mysql S85mysql
# ln -s /etc/init.d/mysql K85mysql
# cd /etc/rc5.d
# ln -s /etc/init.d/mysql S85mysql
# ln -s /etc/init.d/mysql K85mysql
```

安装 **Apache**

```
# mkdir /www
# cd httpd-2.2.14
# ./configure --prefix=/www --enable-so
# make
# make install
```

安装 **php**

```
# mkdir /www/php
# cd php
# ./configure \
# --prefix=/www/php \
# --with-apxs2=/www/bin/apxs \
# --with-libxml-dir=/usr/local/lib \
# --with-zlib \
# --with-zlib-dir=/usr/local/lib \
# --with-gd \
# --with-png-dir=某个目录/libpng-1.2.40 \
# --with-jpeg-dir=某个目录/jpeg-7 \
# --with-mysql=/usr/local/mysql \
# --with-mysqli=/usr/local/mysql/bin/mysql_config \
# --enable-mbstring \
# --enable-soap \
# --enable-sockets
# make
# make install
# cp php.ini-dist /www/php/php.ini
# gedit /www/conf/httpd.conf
在最后加入 AddType application/x-httpd-php .php
```

设置 **apache** 自启动

```
# cp /www/bin/apachectl /etc/init.d/httpd
# cd /etc/rc3.d
# ln -s /etc/init.d/httpd S85httpd
# ln -s /etc/init.d/httpd K85httpd
# cd /etc/rc5.d
# ln -s /etc/init.d/httpd S85httpd
```

```
# ln -s /etc/init.d/httpd K85httpd
```

测试 Apache 和 php

打开浏览器,输入 `http://localhost/`

如出现 `it works`,则 Apache 正常

在 `/www/htdocs` 下建立文件 `test.php`

```
# gedit /www/htdocs/test.php
```

写入:

```
<?php
```

```
phpinfo();
```

```
?>
```

打开浏览器,输入 `http://localhost/test.php`,出现 php 信息(这里显示了 php 的所有配置信息),则说明 PHP 成功.

编译零拷贝模块

```
cd zerocopy/
```

```
make
```

```
insmod ntzc.ko
```

```
rmmod XXXX.ko(网卡驱动)
```

```
insmod XXXX.ko(更改过的网卡驱动)
```

安装 AegisShield

```
# mkdir /etc/snort
```

```
# mkdir /var/log/snort
```

```
# cd snort-2.8.3.1
```

此处,我们提供了一个修改过的 `snort` 源码包,在包中,我们添加了零拷贝模块及应用层协议分析模块,将其改造成了我们的行为分析系统,但是因为以 `snort` 为核心,编译过程相同。

```
# ./configure --with-mysql=/usr/local/mysql
```

```
# make
# make install
#mkdir /var/lib/mysql
# ln -s /tmp/mysql.sock /var/lib/mysql/mysql.sock (这里要建立一个接字的软连接)
```

安装规则

```
# cd 某个目录/ snortrules-snapshot-2[1].8/rules
# cp * /etc/snort
# cd ../etc
# cp snort.conf /etc/snort
# cp *.config /etc/snort
# cp *.map /etc/snort
```

上面用到了两个规则,因为在使用 snortrules-snapshot-2[1].8.tar.gz 解压规则时,

```
include $RULE_PATH/web-client.rules
```

```
include $RULE_PATH/netbios.rules
```

这两个规则编译有问题,所以解压 snortrules-snapshot-CURRENT[1].tar.gz

在/root/so_rules 文件夹下

```
# cp /root/so_rules/netbios.rules /etc/snort
```

```
# cp /root/so_rules/web_client.rules /etc/snort
```

include \$RULE_PATH/mysql.rules 也有同样的问题在/etc/snort/snort.conf 中屏蔽此规则:

```
#include $RULE_PATH/mysql.rules
```

(上面就是把源码包里 rules 文件夹下的内容和 etc 文件下的配置文件复制到 /etc/snort)

安装 web

将系统的 web 界面移动到/www/htdocs/即可。

特别说明:在加载零拷贝模块时,由于网卡驱动多样性,很大的可能性我们的源码包中没有用户系统使用的网卡驱动,此时,用户——主要是指有一定驱动编程能力的程

序员--将不得不自己改写网卡驱动。用户找到自己系统所用的网卡驱动源码之后，可按如下方案更改，使其成为支持零拷贝的网卡驱动（以 Marvell 的网卡驱动 sky2 为例）：

a) 增加 #include "nta.h"

b) struct sk_buff 全部替换为 struct m_buf

c) netdev_alloc_skb(PKT_BUF_SKB) 替换为 nta_alloc_mbuf(NULL, PKT_BUF_SKB, GFP_ATOMIC)

d) skb_reserve 替换为 mbuf_reserve:

e) dev_kfree_skb 替换为 nta_kfree_mbuf:

f) dev_kfree_skb_any 替换为 nta_kfree_mbuf:

g) skb_put 替换为 mbuf_put:

h) skb_frag_t 替换为 mbuf_frag_t

i) skb_shinfo 替换为 mbuf_shinfo

j) skb_copy_to_linear_data 替换为 mbuf_copy_to_linear_data

k) skb_copy_from_linear_data 替换为 mbuf_copy_from_linear_data

l) ip_hdr 换为 ip_header, tcp_hdr 换为 tcp_header

m) ip_hdrlen 换为 ip_headerlen

n) skb_fill_page_desc 换为 mbuf_fill_page_desc

o) sky2_xmit_frame 拆解为两个函数 sky2_xmit_frame 和 sky2_xmit_frame_fake，并且.ndo_start_xmit = sky2_xmit_frame_fakep)

添加 nta_register_zc(dev, pcnet32_start_xmit);

基本修改方案即如上所述,本质上其实是将所有的 sk_buff 的操作替换为 m_buf 的操作，在此基础上编译,若出现错误,只要按此原则顺序解决即可,不会存在不可解决的障碍,此时,网卡驱动已经准备好了。


4.4 测试用例及结果

4.4.1 应用层协议分析模块

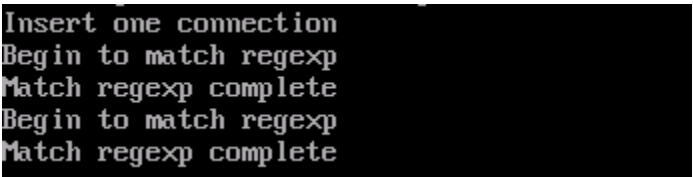
子模块：规则文件管理模块

测试目标	规则文件被正确编译并存入链表
测试方法	将协议名输入本地文件/etc/snort/proto.config 将协议文件放入本地文件夹/etc/snort/regexps 启动 snort 查看正则表达式是否被正确载入
测试结果	<p>正则表达式被正确载入</p> <p>后台监控</p> <p>本地正则表达式载入</p> <pre>AddPatternFromFile: /etc/snort/regexps/edonkey.pat 3 success AddPatternFromFile: /etc/snort/regexps/bittorrent.pat 4 success AddPatternFromFile: /etc/snort/regexps/msnmessenger.pat 5 success AddPatternFromFile: /etc/snort/regexps/kugoo.pat 6 success AddPatternFromFile: /etc/snort/regexps/yahoo.pat 7 success AddPatternFromFile: /etc/snort/regexps/ftp.pat 8 success AddPatternFromFile: /etc/snort/regexps/http.pat 9 success AddPatternFromFile: /etc/snort/regexps/pop3.pat 10 success AddPatternFromFile: /etc/snort/regexps/pplive.pat 11 success AddPatternFromFile: /etc/snort/regexps/smtp.pat 12 success AddPatternFromFile: /etc/snort/regexps/ssh.pat 13 success AddPatternFromFile: /etc/snort/regexps/telnet.pat 14 success</pre> <p>Web 端添加的正则表达式载入</p> <pre>Begin to read updateFile AddPatternFromFile: /etc/snort/update_regexps/qq.pat 15 success AddPatternFromFile: /etc/snort/update_regexps/xunlei.pat 16 success Read updateFile complete</pre>
结果分析	测试通过 [√]; 测试未通过 □ []

子模块：连接管理模块

测试目标	连接被正确插入和删除
测试方法	启动 snort 客户机进行各种网络行为 观察连接插入删除情况
测试结果	连接被正确插入和删除 后台监控 连接被正确插入和删除 
结果分析	测试通过 <input checked="" type="checkbox"/> [√]; 测试未通过 <input type="checkbox"/> []

子模块：规则匹配模块

测试目标	数据包是否匹配正则表达式
测试方法	启动 snort 客户机进行各种网络行为 观察匹配情况
测试结果	数据包匹配正则表达式 后台监控 数据包被匹配 
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

子模块：输出过滤模块

测试目标	数据是否被缓冲并输出
测试方法	启动 snort 客户机进行各种网络行为 观察过滤情况
测试结果	连续的数据包被过滤
结果分析	测试通过 <input checked="" type="checkbox"/> [√]; 测试未通过 <input type="checkbox"/> []

子模块：输出模块

测试目标	数据是否被缓冲并输出
测试方法	<p>启动 snort</p> <p>客户机进行各种网络行为</p> <p>观察缓冲和输出情况</p>
测试结果	<p>数据包被缓冲并且输出</p> <p>后台监控</p> <p>数据包被缓冲并且输出</p> <pre>source: 192.168.1.106:4000 proto: qq write success begin swapping swap complete Insert into success</pre> <pre> 13 1277017528 1778493632 40975 qq</pre>
结果分析	<p>测试通过 <input checked="" type="checkbox"/> [√]; 测试未通过 <input type="checkbox"/> []</p>

主模块：应用层协议匹配模块

验证协议匹配一（QQ2010 Beta 1530）

测试目标	检测出使用 QQ 协议的主机 IP 和端口号
测试方法	启动 snort 客户机登录 QQ 观察输出情况
测试结果	<p>使用 QQ 的主机被检测到并输出信息到数据库</p> <p>后台监控</p> <pre>source: 192.168.1.106:4000 proto: qq write success begin swapping swap complete Insert into success</pre> <pre>13 1277017528 1778493632 40975 qq</pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

验证协议匹配二（迷你迅雷 3.0 build: 3.1.1.58）

测试目标	检测出使用迅雷协议的主机 IP 和端口号
测试方法	<p>启动 snort</p> <p>客户机登录迅雷</p> <p>3) 观察输出情况</p>
测试结果	<p>使用迅雷的主机被检测到并输出信息到数据库</p> <p>后台监控</p> <pre>source: 192.168.1.106:53378 proto: xunlei write success begin swapping swap complete Insert into success</pre> <pre> 14 1277017872 1778493632 33488 xunlei</pre>
结果分析	<p>测试通过 <input checked="" type="checkbox"/> [√]; 测试未通过 <input type="checkbox"/> []</p>

验证协议匹配三（edonkey 1.1.7）

测试目标	检测出使用 edonkey 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机登录 edonkey 3) 观察输出情况
测试结果	<p>使用 edonkey 的主机被检测到并输出到数据库</p> <p>后台监控</p> <pre>source: 192.168.1.106:53406 proto: edonkey write success begin swapping swap complete Insert into success</pre> <pre> 15 1277018029 1778493632 40656 edonkey </pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

验证协议匹配四（windows live messenger 版本 14.0.8089.726）

测试目标	检测出使用 msn 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机登录 msn 3) 观察输出情况
测试结果	使用 msn 的主机被检测到并输出到数据库 后台监控 <pre> source: 192.168.1.106:53415 proto: msnmessenger write success begin swapping swap complete Insert into success </pre> <pre> ! 16 ! 1277018186 ! 1778493632 ! 42960 ! msnmessenger </pre>
结果分析	测试通过 <input type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

验证协议匹配五（ftp）

测试目标	检测出使用 ftp 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机登录某个 ftp 3) 观察输出情况
测试结果	使用 ftp 的主机被检测到并输出到数据库 后台监控 <pre> source: 192.168.1.106:53597 proto: ftp write success begin swapping swap complete Insert into success </pre> <pre> 17 1277018513 1778493632 24017 ftp </pre>
结果分析	测试通过 <input checked="" type="checkbox"/> [√] ; 测试未通过 <input type="checkbox"/> []□

验证协议匹配六（http）

测试目标	检测出使用 http 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机登录浏览器，打开任意网页 3) 观察输出情况
测试结果	<p>使用 http 的主机被检测到并输出到数据库</p> <p>后台监控</p> <pre>source: 192.168.1.106:53615 proto: http write success begin swapping swap complete Insert into success</pre> <pre> 18 1277018631 1778493632 28625 http</pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ； 测试未通过 <input type="checkbox"/>

验证协议匹配七（ssh）

测试目标	检测出使用 ssh 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机使用 ssh 3) 观察输出情况
测试结果	使用 ssh 的主机被检测到并输出到数据库 后台监控 <pre> source: 192.168.1.106:53742 proto: ssh write success begin swapping swap complete Insert into success </pre> <pre> ! 19 ! 1277018946 ! 1778493632 ! 61137 ! ssh </pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

验证协议匹配八（kugoo build: 6.1.18.404）

测试目标	检测出使用 kugoo 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机使用 kugoo 3) 观察输出情况
测试结果	使用 kugoo 的主机被检测到并输出到数据库 后台监控 <pre>source: 192.168.1.106:5041 proto: kugoo write success begin swapping swap complete Insert into success</pre> <pre> 20 1277022515 1778493632 45331 kugoo</pre>
结果分析	测试通过 <input checked="" type="checkbox"/> [√]; 测试未通过 <input type="checkbox"/> []□

验证协议匹配九（dns）

测试目标	检测出使用 dns 协议的主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机使用 dns 进行地址解析 3) 观察输出情况
测试结果	使用 dns 的主机被检测到并输出到数据库 后台监控 <pre> source: 192.168.1.106:61340 proto: dns write success begin swapping swap complete Insert into success </pre> <pre> : 21 1277022865 1778493632 40175 dns </pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

验证协议匹配十（bittorrent 6.4 build 18095）

测试目标	检测出使用 bittorrent 主机 IP 和端口号
测试方法	1) 启动 snort 2) 客户机使用 bittorrent 3) 观察输出情况
测试结果	使用 bittorrent 的主机被检测到并输出到数据库 后台监控 <pre> source: 192.168.1.106:35754 proto: bittorrent write success begin swapping swap complete Insert into success </pre> <pre> ! 26 ! 1277023445 ! 1778493632 ! 43659 ! bittorrent </pre>
结果分析	测试通过 <input checked="" type="checkbox"/> ; 测试未通过 <input type="checkbox"/>

4.4.2 零拷贝部分

测试目标	测试 ntzc 模块的正常加载
测试说明	零拷贝部分采用 LKM 的方法，须手动加载到内核，才可完成零拷贝的支持。其中包括重要的内存管理部分，及网卡驱动部分，而网卡驱动依赖于内存管理模块，后者必须先于前者加载到内核中。同时，运行零拷贝的机器的网卡驱动必须采用模块形式挂在到内核上，而非集成，才有可能完成网卡驱动的替换。
测试环境	硬件：CPU：intel T5450 @1.66 Ghz；内存：2G；硬盘：160G；网卡：Marvell 88E8039(有线)，Intel Corporation PRO/Wireless 3945ABG（无线） 软件：系统：Ubuntu 10.04；kernel：2.6.32；gcc 4.4.3
测试方法	1) 切换到 root 用户（或者使用 sudo 代替） 2) 加载 ntzc 模块—insmod ntzc.ko 3) 卸载原有网卡驱动—rmmod XXXX.ko(对于测试机来说为 sky2.ko) 4) 加载支持零拷贝的网卡驱动—insmod XXXX.ko
测试结果	<pre> zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ ls bnx2.c bvl.c Makefile nta.o pcnet32.c sky2.mod.c zc.o bnx2_fw2.h bvl.h modules.order ntzc.ko README sky2.mod.o bnx2_fw.h bvl.o Module.symvers ntzc.mod.c sky2.c sky2.o bnx2.h igb nta.c ntzc.mod.o sky2.h zc.c built-in.o ixgbe nta.h ntzc.o sky2.ko zc_comm.h zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ sudo insmod ntzc.ko zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ sudo rmmod sky2 ERROR: Module sky2 does not exist in /proc/modules zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ sudo insmod sky2.ko zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ lsmod grep sky2 sky2 39545 0 ntzc 15321 1 sky2 zhangwei@zhangwei-ubuntu:~/programme/zerocopy/zc\$ </pre>
结果分析	测试通过，ntzc 可正常加载

测试目标	测试零拷贝抓包
测试说明	此部分采用 ntzc 自带的抓包程序 sniff 来抓取数据包，当然在测试之前 ntzc.ko 和 sky2.ko（测试机网卡驱动）已经成功加载进内核
测试环境	硬件：CPU: intel T5450 @1.66 Ghz；内存：2G；硬盘：160G；网卡：Marvell 88E8039(有线)，Intel Corporation PRO/Wireless 3945ABG（无线） 软件：系统：Ubuntu 10.04；kernel：2.6.32；gcc 4.4.3
测试方法	1) 切换到 root 用户（或者使用 sudo 代替） 2) cd zerocopy/nta 3) 运行 ./sniff -i eth0
测试结果	<pre> zc_handle test packet: 00 00 00 0c 00 1f 3c 36 94 4c 86 dd 60 00 00 00 00 9a 11 01 fe 80 00 00 00 00 00 00 21 e0 c9 bf c4 cc ab 5b ff 02 00 00 00 00 00 00 00 00 00 00 00 00 00 0c da af 07 6c 00 9a 60 e7 4d 2d 53 45 41 52 43 48 20 2a 20 48 54 54 50 2f 31 2e 31 0d 0a 48 6f 73 74 3a 5b 46 46 30 32 3a 3a 43 5d 3a 31 39 30 30 0d 0a 53 54 3a 75 72 6e 3a 4d 69 63 72 6f 73 6f 66 74 20 57 69 6e 64 6f 77 73 20 50 packet:3, snaplen:128 zc_handle test packet: 33 33 ff dd a9 81 00 1f 3c 36 94 4c 86 dd 60 00 00 00 00 20 3a ff fe 80 00 00 00 00 00 00 21 e0 c9 bf c4 cc ab 5b ff 02 00 00 00 00 00 00 00 00 00 01 ff dd a9 81 87 00 ad bf 00 00 00 00 fe 80 00 00 00 00 00 00 29 d6 6b df b8 dd a9 81 01 01 00 1f 3c 36 94 4c packet:4, snaplen:86 zc_handle test packet: 33 33 ff dd a9 81 00 1f 3c 36 94 4c 86 dd 60 00 00 00 00 20 3a ff fe 80 00 00 00 00 00 00 21 e0 c9 bf c4 cc ab 5b ff 02 00 00 00 00 00 00 00 00 00 01 ff dd a9 81 87 00 ad bf 00 00 00 00 fe 80 00 00 00 00 00 00 29 d6 6b df b8 dd a9 81 01 01 00 1f 3c 36 94 4c packet:5, snaplen:86 </pre>
结果分析	测试通过，自带的 sniff 程序可以抓到数据包，与此同时，使用 wireshark 或 tcpdump 等不能抓到数据包，这证明抓取的数据包确实是存储在自定义缓冲区中，零拷贝模式能够正常抓包。

测试目标	测试零拷贝的抓包性能（1）
测试说明	此部分主要是测试零拷贝相对普通模式的抓包性能有多大提高。
测试环境	硬件：虚拟机 Vmware Fusion；CPU：双核； 内存：4G； 软件：系统: CentOS ;kernel:2.6.32； gcc 4.4.3
测试方法	使用 stick（snort 的专用压力测试工具）不断向目标机发送数据包 目标机单独运行零拷贝部分（未涉及到 snort），使用自带的测试程序 --sniff（在 nta 中）抓包， 读报文头然后释放，计数。
测试结果	虚拟机上每秒可以做到 $250 \times 8k = 2M$ pps（2K 以内各种大小报文对性能没有影响）；sniff 程序,大概占用了一个核心处理能力的 10-20%；
结果分析	见下。

测试目标	测试零拷贝的抓包性能（2）
测试说明	此部分主要是测试零拷贝相对普通模式的抓包性能有多大提高。
测试环境	硬件：Intel 2.0G CPU, 4 核 1M cache, BCM 万兆卡 软件：系统: CentOS ;kernel:2.6.32; gcc 4.4.3
测试方法	1) 使用 stick (snort 的专用压力测试工具) 不断向目标机发送数据包 2) 直接修改原生驱动程序在 netif_receive 之前释放收到的报文，计数，观察数据。 运行 ntzc+sniff，抓包之后释放，计数，观察数据。
测试结果	1) 没有运行 sniff 之前，直接修改原生驱动程序在 netif_receive 之前释放收到的报文，CPU1 个核可以达到 1.7M pps 的收包能力，再大就会丢包； 2) 运行 ntzc+sniff 后，CPU 可以达到 1.1M pps 的抓包能力（包括用户空间 sniffer 收包、计数、读报文头和释放空间），此时 CPU1 个核被 ksoftirq 线程占满，另一个核运行 sniff 程序，CPU 占用率 <30%；
结果分析	在运行上一测试用例及本测试用例的过程中，发现 ntzc 本身并没有增强对突发小包的捕获能力，但大大节省了大包到用户空间时 memcpy 对 CPU 的占用；另外，由于 ntzc 可以很容易支持每个核上独立管理报文内存，如果配合网卡的多通道能力，在多进（线）程用户程序处理报文时，可以最大程度减少 cache 在多个核间同步带来的性能损失。

测试目标	测试添加了零拷贝支持的 snort 的编译
测试说明	这部分需要测试 snort 与零拷贝模块的集成情况，snort 必须能够使用零拷贝的抓包模块，才算真正的完成。
测试环境	硬件：CPU: intel T5450 @1.66 Ghz；内存：2G；硬盘：160G；网卡：Marvell 88E8039(有线)，Intel Corporation PRO/Wireless 3945ABG（无线） 软件：系统：Ubuntu 10.04；gcc 4.4.3
测试方法	1) 进入 snort 的安装包（snort 源码已经改造完成） 2) ./configure --with-mysql=/usr/local/mysql 3) make 4) make install
测试结果	编译通过，无错误出现
结果分析	添加了零拷贝的模块能够成功编译，snort 与零拷贝整合良好

测试目标	测试支持零拷贝的 snort 嗅探模式运行情况
测试说明	这部分将初步测试 snort 与零拷贝模块集成之后的运行情况，snort 必须能够使用零拷贝的抓包模块，才算真正的完成。
测试环境	<p>硬件：CPU: intel T5450 @1.66 Ghz；内存：2G；硬盘：160G；网卡：Marvell 88E8039(有线)，Intel Corporation PRO/Wireless 3945ABG（无线）</p> <p>软件：系统：Ubuntu 10.04；gcc 4.4.3</p>
测试方法	<p>1) snort 编译通过</p> <p>2) 运行 ./snort -i eth0 -v</p>

测试结果	<pre>-- Initializing snort -- Initializing Output Plugins! Verifying Preprocessor Configurations! Initializing Network Interface eth0 Using the ZERO-COPY functions! Init ring_num 32704 entry_num = 41 mmap: 0.41: cpu: 0, ptr: 0xb7314000, number: 128, order: 2, offset: 0. mmap: 1.41: cpu: 0, ptr: 0xb7114000, number: 128, order: 2, offset: 512. mmap: 2.41: cpu: 0, ptr: 0xb6f14000, number: 128, order: 2, offset: 1024. mmap: 3.41: cpu: 0, ptr: 0xb6d14000, number: 128, order: 2, offset: 1536. mmap: 4.41: cpu: 0, ptr: 0xb6b14000, number: 128, order: 2, offset: 2048. mmap: 5.41: cpu: 0, ptr: 0xb6914000, number: 128, order: 2, offset: 2560. mmap: 6.41: cpu: 0, ptr: 0xb6714000, number: 128, order: 2, offset: 3072. mmap: 7.41: cpu: 0, ptr: 0xb6514000, number: 128, order: 2, offset: 3584. mmap: 8.41: cpu: 0, ptr: 0xb6314000, number: 128, order: 2, offset: 4096. mmap: 9.41: cpu: 0, ptr: 0xb6114000, number: 128, order: 2, offset: 4608. mmap: 10.41: cpu: 0, ptr: 0xb5f14000, number: 128, order: 2, offset: 5120. mmap: 11.41: cpu: 0, ptr: 0xb5d14000, number: 128, order: 2, offset: 5632. mmap: 12.41: cpu: 0, ptr: 0xb5b14000, number: 128, order: 2, offset: 6144. mmap: 13.41: cpu: 0, ptr: 0xb5914000, number: 128, order: 2, offset: 6656. mmap: 14.41: cpu: 0, ptr: 0xb5714000, number: 128, order: 2, offset: 7168. mmap: 15.41: cpu: 0, ptr: 0xb5514000, number: 128, order: 2, offset: 7680.</pre> <pre>Read updateFile complete 06/29-20:37:28.657597 ARP who-has 192.168.0.1 tell 192.168.0.103 06/29-20:38:03.667544 ARP who-has 192.168.0.1 tell 192.168.0.103 06/29-20:38:17.125858 ARP who-has 192.168.0.100 tell 192.168.0.102 06/29-20:38:17.645258 192.168.0.103 -> 224.0.0.22 IGMP TTL:1 TOS:0x0 ID:10289 IpLen:24 DgmLen:40 IP Options (1) => RTRALT ==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+ 06/29-20:38:17.662115 192.168.0.103 -> 224.0.0.22 IGMP TTL:1 TOS:0x0 ID:10290 IpLen:24 DgmLen:40 IP Options (1) => RTRALT ==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+ 06/29-20:38:17.671481 192.168.0.103 -> 224.0.0.22 IGMP TTL:1 TOS:0x0 ID:10291 IpLen:24 DgmLen:40 IP Options (1) => RTRALT ==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+==+=+</pre>
结果分析	由第一个图可以看出，snort 的零拷贝模式成功启动；第二个图中，snort 在零拷贝模式下成功嗅探到了网络数据包。测试通过。

测试目标	测试支持零拷贝的 snort 的 IDS 模式运行情况
测试说明	即使已经证明 snort 嗅探模式无误，仍需验证其 IDS 模式，因为此功能才是本系统最看重的能力
测试环境	硬件：CPU: intel T5450 @1.66 Ghz；内存：2G；硬盘：160G；网卡：Marvell 88E8039(有线)，Intel Corporation PRO/Wireless 3945ABG（无线） 软件：系统：Ubuntu 10.04；gcc 4.4.3
测试方法	1) snort 编译通过 2) 运行 ./snort -i eth0 -c /etc/snort/snort.conf
测试结果	<pre> mmmap: 37.41: cpu: 0, ptr: 0xb1d62000, number: 128, order: 2, offset: 18944. mmmap: 38.41: cpu: 0, ptr: 0xb1b62000, number: 128, order: 2, offset: 19456. mmmap: 39.41: cpu: 0, ptr: 0xb1962000, number: 128, order: 2, offset: 19968. mmmap: 40.41: cpu: 0, ptr: 0xb1762000, number: 128, order: 2, offset: 20480. mmmap ring zone: zcb[0] 0xb1763000_zr[0] 0xb1762000 entry 0xb1762000 mmmap ring zone: zcb[1] 0xb1762c00_zr[1] 0xb1762000 entry 0xb1762000 OpenPcap() device eth0 network lookup: eth0: no IPv4 address assigned Decoding Ethernet on interface eth0 database: compiled support for (mysql) database: configured to use mysql database: user = root database: password is set database: database name = snort database: host = localhost database: sensor name = unknown:eth0 database: sensor id = 3 database: schema version = 107 database: using the "log" facility [Port Based Pattern Matching Memory] --[AC-BNFA Search Info Summary]----- Instances : 880 Patterns : 177618 Pattern Chars : 1777082 Num States : 1347400 Num Match States : 190548 Memory : 29.52Mbytes Patterns : 5.08M Match Lists : 8.12M Transitions : 16.12M ----- --== Initialization Complete ==-- --> Snort! <*- o")~ Version 2.8.3.1 (Build 17) "" By Martin Roesch & The Snort Team: http://www.snort.org/team.html </pre>

	<pre>Breakdown by protocol (includes rebuilt packets): ETH: 16231 (100.000%) ETHdisc: 0 (0.000%) VLAN: 0 (0.000%) IPV6: 81 (0.499%) IP6 EXT: 0 (0.000%) IP6opts: 0 (0.000%) IP6disc: 0 (0.000%) IP4: 16050 (98.885%) IP4disc: 0 (0.000%) TCP 6: 0 (0.000%) UDP 6: 0 (0.000%) ICMP6: 0 (0.000%) ICMP-IP: 0 (0.000%) TCP: 15360 (94.634%) UDP: 557 (3.432%) ICMP: 3 (0.018%) TCPdisc: 0 (0.000%) UDPdisc: 0 (0.000%) ICMPdis: 0 (0.000%) FRAG: 0 (0.000%) FRAG 6: 0 (0.000%) ARP: 5 (0.031%) EAPOL: 0 (0.000%) ETHLOOP: 0 (0.000%) IPX: 0 (0.000%) OTHER: 95 (0.585%) DISCARD: 0 (0.000%) InvChkSum: 0 (0.000%) S5 G 1: 0 (0.000%) S5 G 2: 130 (0.801%) Total: 16231 ===== Action Stats: ALERTS: 3 LOGGED: 3 PASSED: 0 =====</pre>
结果分析	由第一个图可以看出，snort 的 IDS 模式运行无误；第二个图中，snort 在零拷贝 IDS 模式下成功抓取到网络数据包。测试通过。

4.5 多核调度分析

流水线中各节点的复杂度分析

节点名称	处理时间	备注
抓取数据包	1T	指使用零拷贝技术后,从数据到达网卡并直接送到用户空间所需要的时间。以该处理时间作为衡量其他节点的单位。
预处理	11T	指在使用 flag3、Stream5、http、rpc、bo、ftp、telnet、smtp、portscan、arpspoof等预处理插件下,所需的时间。
入侵检测分析	57T	指在使用 2009年10月20日为止的 Snort 规则库下,检测一个包平均的时间。
应用层行为分析	14T	指在使用本系统自定义应用层行为分析,检测一个数据包所需的时间。
输出	6T	指从 DetectServer 把一条数据写入到 WebServer 所需的时间。

*上述处理时间,为在 Intel T2350 处理器下,1000M 网络环境下的实测结果。

结论分析:

利用手动调度均衡 CPU 负荷,可以估算出各个节点占用 CPU 资源的计算式子,其在最理想状态下的表达式为:

$$CPU_s = CPU_{Part1} + CPU_{Part2} + CPU_{Part3} + CPU_{Part4} + CPU_{Part5}$$

其中

$$CPU_{Part1} = (1/89) * CPU_s$$

$$CPU_{Part2} = (11/89) * CPU_s$$

$$CPU_{Part3} = (57/89) * CPU_s$$

$$CPU_{Part4} = (14/89) * CPU_s$$

$$CPU_{Part5} = (6/89) * CPU_s$$

*公式中暂时忽略了系统其他应用程序与系统进程所需的 CPU 资源，默认 DetectServer 当中只运行 Aegis Sheild。

注：CPUs 指总的 CPU 资源，CPU_{Partk} 指第 k 个节点所占用的 CPU 资源。
对应的节点如下表：

简写	节点
Part1	抓取数据包
Part2	预处理
Part3	入侵检测分析
Part4	应用层行为分析
Part5	输出

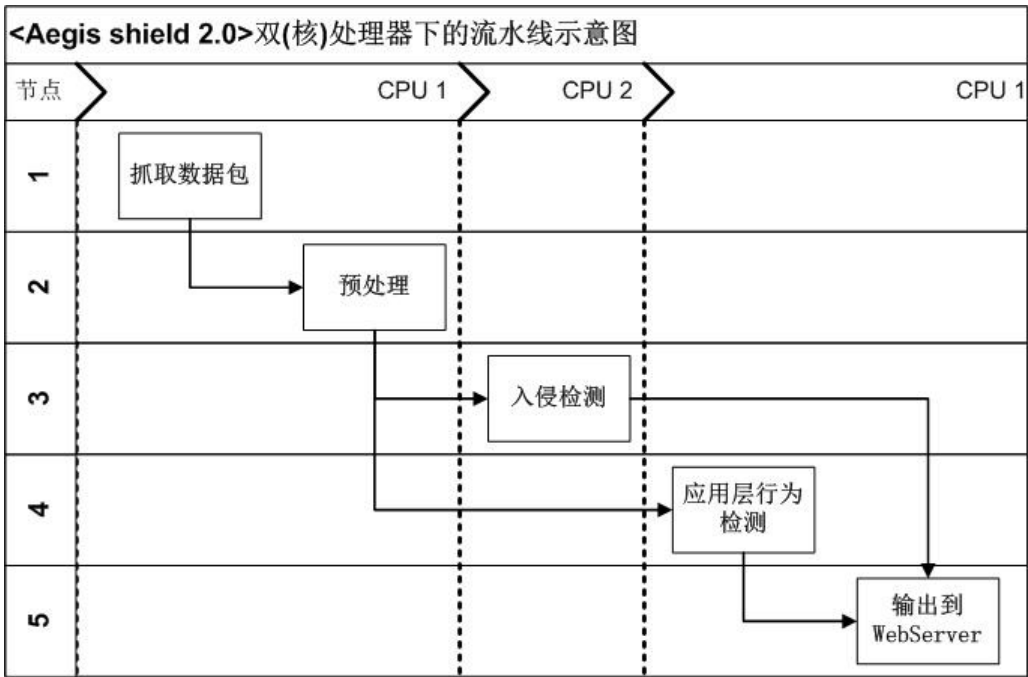
多(核)处理器下的调度策略

双处理器的调度策略

根据表达式有：

$$CPU_{Part3} > CPU_{Part1} + CPU_{Part2} + CPU_{Part4} + CPU_{Part5}$$

因此，可把整体划分为两个部分，划分后的流水线如下：

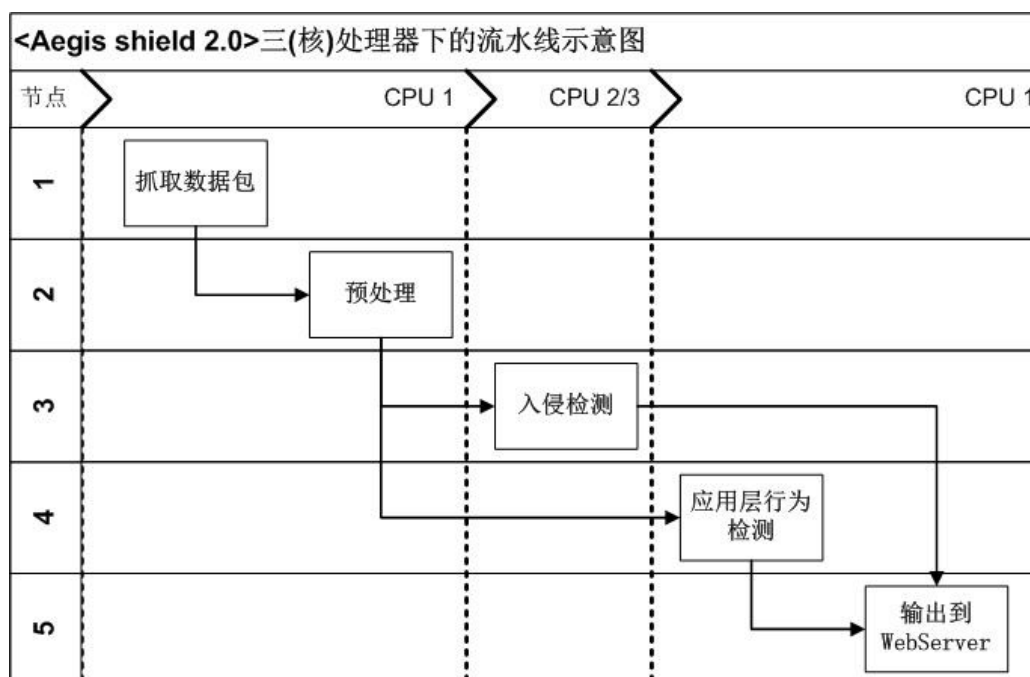


三处理器的调度策略

根据表达式有：

$$(1/2)CPU_{Part3} \approx CPU_{Part1} + CPU_{Part2} + CPU_{Part4} + CPU_{Part5}$$

因此，可以由两个处理器同时运行 Part3 部分，如双处理器的划分类似，划分后的流水线如下：



N 处理器下的调度策略

相关结论：

根据表达式，有如下几个结论

$$CPU_{Part1} + CPU_{Part4} \approx CPU_{Part2} + CPU_{Part5} \approx (1/4) CPU_{Part3}$$

比较均衡的程况下，要求有六个或六倍数个处理器并行执行。

但在实际情况下， CPU_{Part1} 与 CPU_{Part5} 是不可多处理并行部分。

因为 CPU_{Part1} 是抓取数据包的过程，必须从驱动的底层上进行并行优化，考虑系统稳定性以及 CPU_{Part1} 上花费时间，因此可以在实际分配 CPU 中，忽略。

而 CPU_{Part5} 的时间，主要由网络的带宽与 WebServer 的数据库性能决定，在 DetectServer 处作缓存优化能从数量级上降低一个检测出的数据包输出平所需的时间，在实际的 CPU 分配当中也可以忽略不计，让 SMP 进行处理。

因此，可以得出多处理器下实际的分配均衡策略表达式：

$$\text{CPU}_{\text{Part2}} \approx \text{CPU}_{\text{Part3}} \approx (1/5)\text{CPU}_{\text{Part4}}$$

或

$$\text{CPU}_{\text{Part2}} + \text{CPU}_{\text{Part3}} \approx (1/2)\text{CPU}_{\text{Part4}}$$

上述两个表达式中，当 N 的数量足够大（如 $N > 7$ ）时，使用表达式一较为合理，因为可以减少进程切换花费的不必要开销。

调度策略：

设处理器的数量为 K

当 $K < 4$ 时，遵循 $k=2$ 或 $k=3$ 时的策略

当 $K \geq 4$ 时，

（注，下式中，左值代表各节点的 CPU 资源，右值代表使用的 CPU，考虑到进程切换，下面将分配的 CPU 进行具体的取整策略）

$K < 7$ 时，

（注，右值将以四舍五入方式取整数）

$$\text{CPU}_{\text{Part2}} + \text{CPU}_{\text{Part3}} = (1/3)\text{CPUs}$$

$$\text{CPU}_{\text{Part4}} = (2/3) \text{ CPUs}$$

$K \geq 7$ 时，

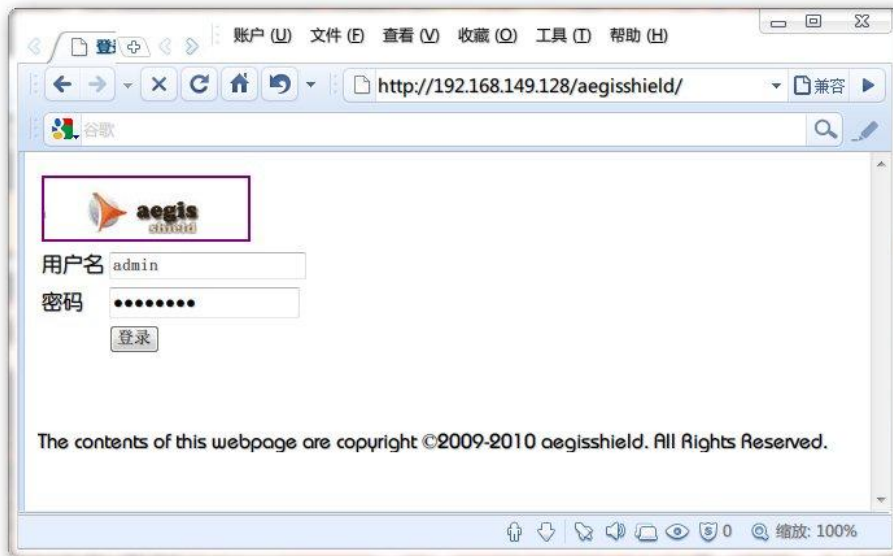
（注， $\text{CPU}_{\text{Part2}}$ 与 $\text{CPU}_{\text{Part3}}$ 将直接取整， $\text{CPU}_{\text{Part4}}$ 值将以四舍五入方式取整数）

$$\text{CPU}_{\text{Part2}} = \text{CPU}_{\text{Part3}} = (1/7)\text{CPUs}$$

$$\text{CPU}_{\text{Part4}} = (5/7) \text{ CPUs}$$

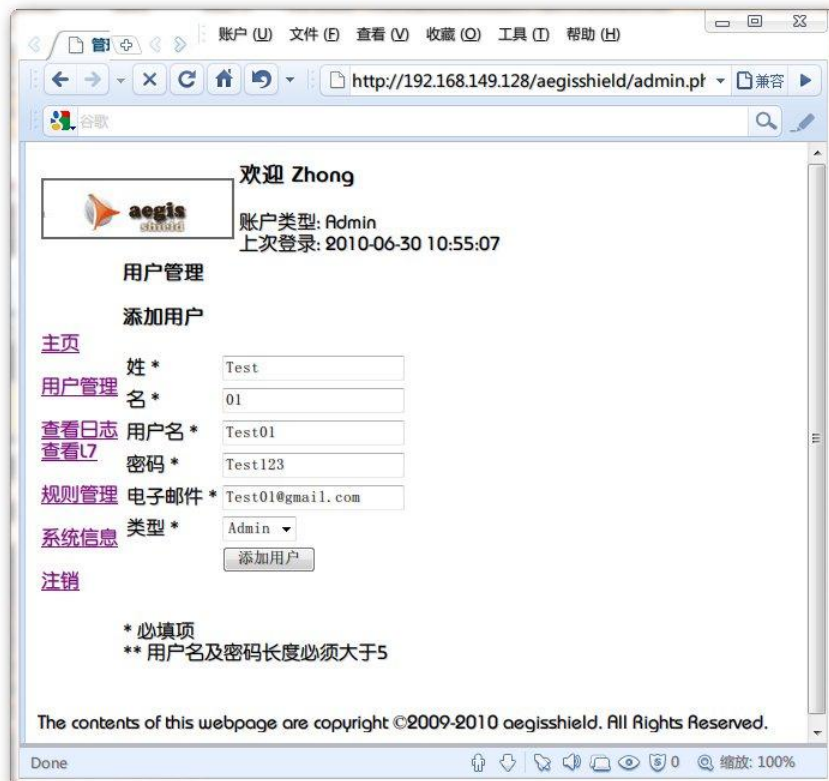
4.6 WEB 功能测试

说明：WEB 功能测试，只对部分功能进行测试，未覆盖全部可选项条件
登陆



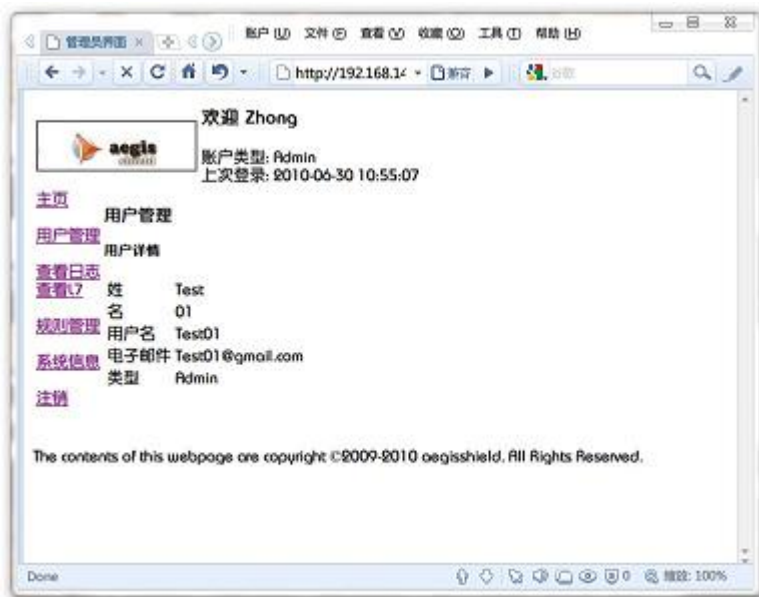
用户管理

admin 管理员账号登陆进行用户管理的添加用户操作

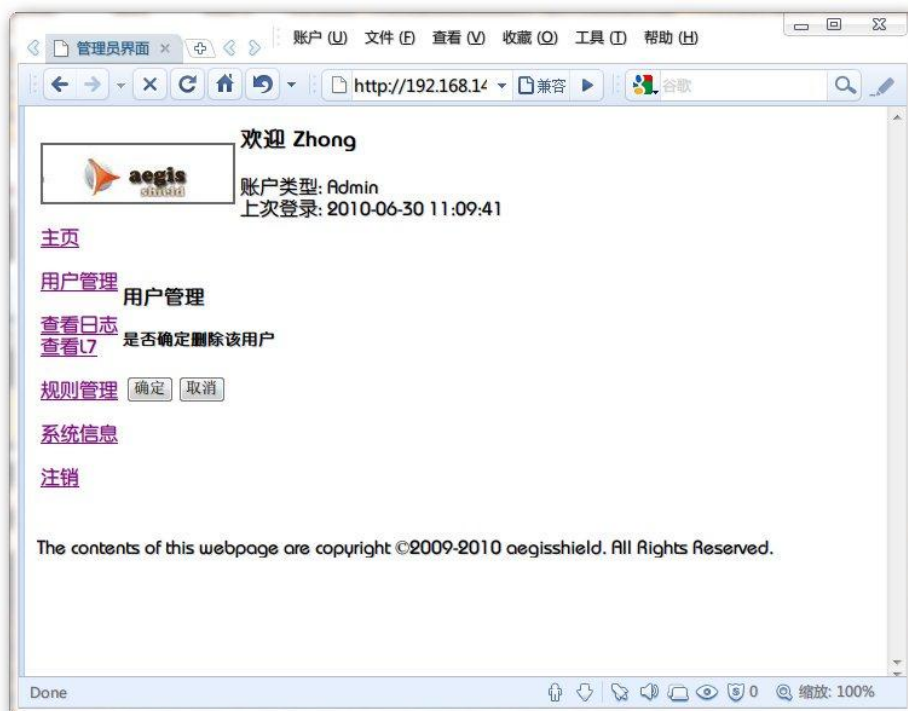




查看用户



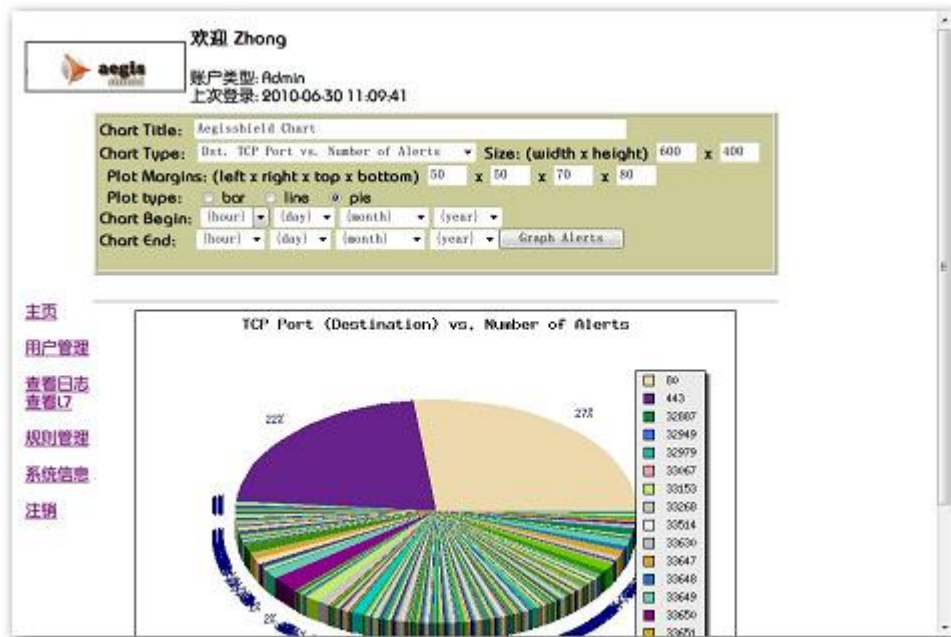
删除用户操作



查看日志

(这里展示各种方式查看日志)

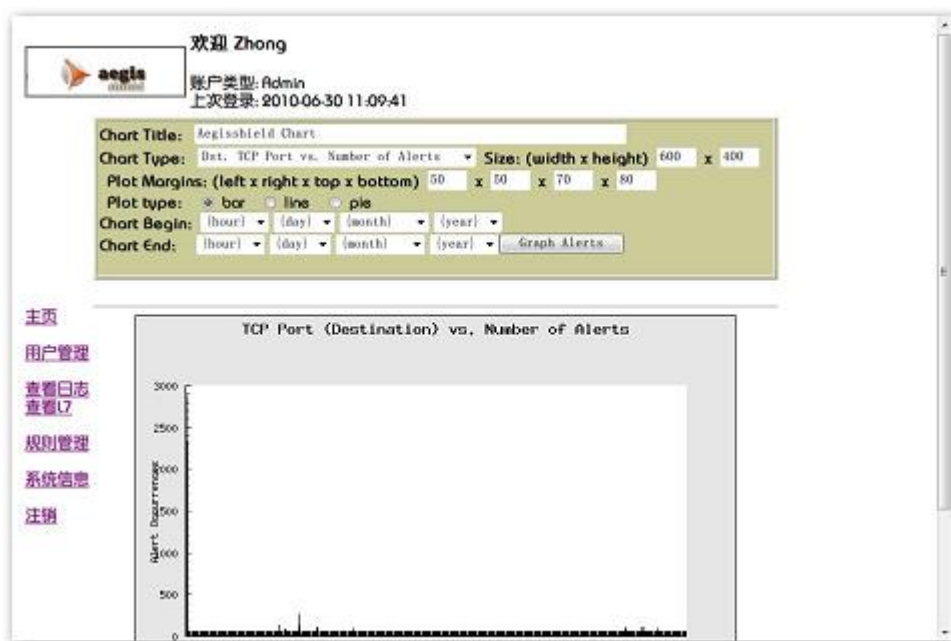
TCP 目的端口号与报警数关系的饼状图表示



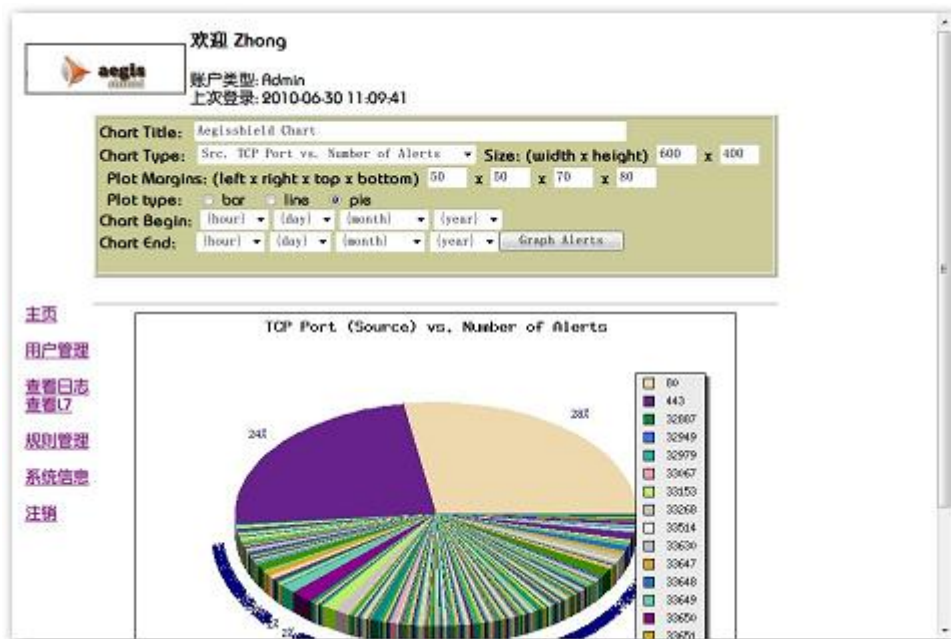
TCP 目的端口号与报警数关系的线性图表示



TCP 目的端口号与报警数关系的柱状图表示



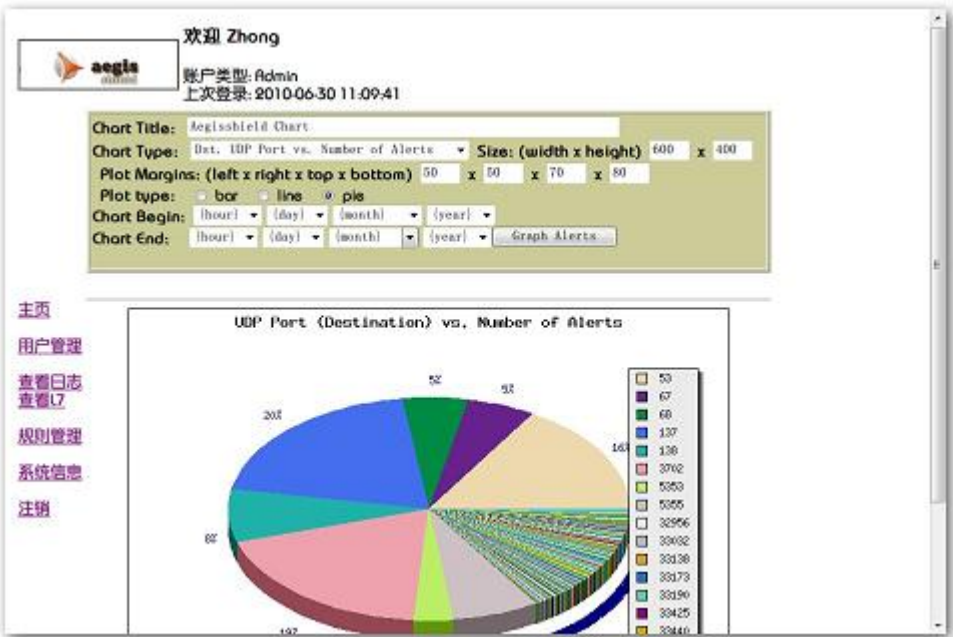
TCP 源端口号与报警数关系的饼状图表示



TCP 源端口号与报警数关系的线性图表示



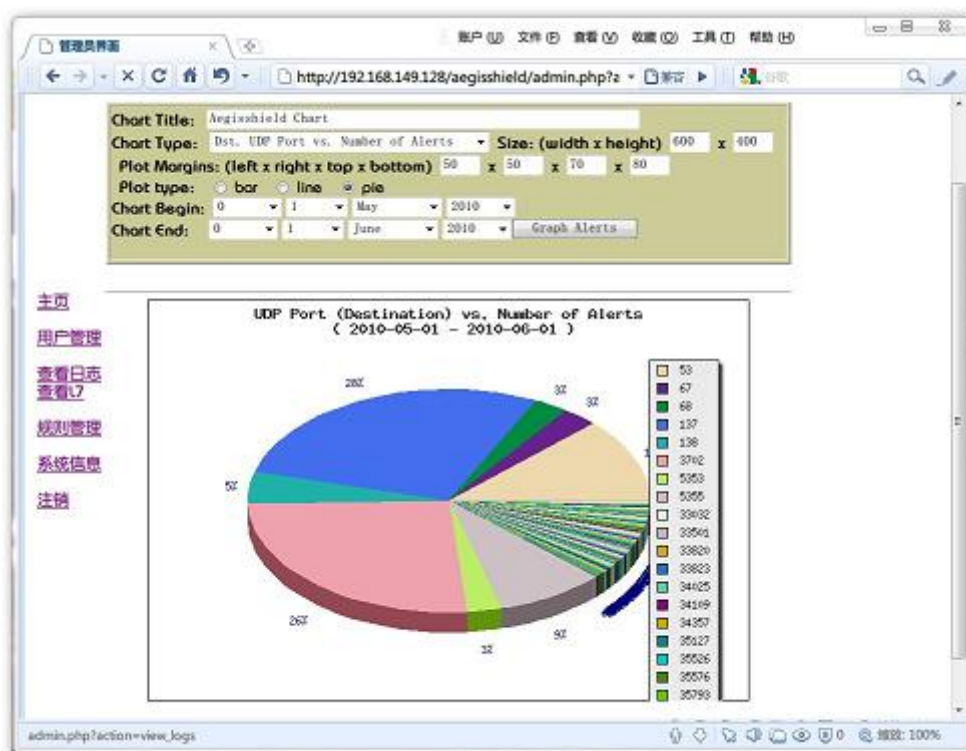
UDP 目的端口号与报警数关系的饼状图表示



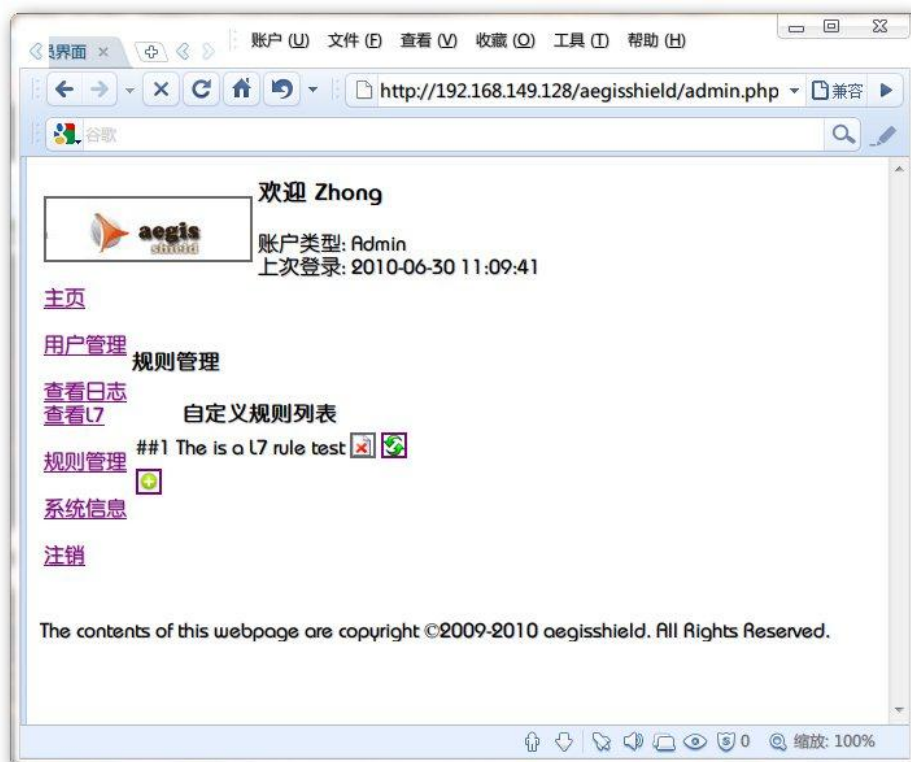
UDP 目的端口号与报警数关系的线性图表示



某段时间类的某种类型图表显示



规则管理







一般用户登陆页面



第五章 创新性

相对于传统的网络行为分析系统，我们的系统具有以下几个亮点：

1) 使用零拷贝技术

系统支持零拷贝模式和普通模式自由切换。千兆或更高的网络下，零拷贝模式可以减少数据拷贝和共享总线操作的次数，消除通信数据在存储器之间不必要的中间拷贝过程，降低系统调用的开销，有效地提高抓取数据包的效率。而对普通模式的支持可以增强系统的可靠性和健壮性。

2) 使用多核编程技术

充分发挥现有多核处理器的能力，系统同一时刻能够并行处理多个数据包，大大提高了本行为分析系统的工作效率，弥补了现有入侵检测系统等因串行处理不及而丢失数据包的问题；

3) 应用层行为分析

通过使用应用层传输数据特征库，快速识别出多达十余种国内最常见的网络软件，如 QQ、P2P、HTTP 等，从而更加有效的帮助管理员监控网络流量及对网内用户进行管理。

4) 基于 B/S 的友好界面

通过友好简易的操作界面，网络管理员只需要简单地培训，就可以有效地使用。更重要的是，管理员可以在任意有浏览器支持的平台上登录 WEB 服务器操作。我们使用了分级 WEB 管理，以适应不同的使用角色；

第六章 总结

当今的时代是网络的时代，网络的普及率越来越高，应用面越来越广，与之伴随的是网络的安全性越来越受到人们的重视。防火墙，入侵检测系统，数据认证等等，一系列手段都在努力保障人们的信息安全，即使这样，我们的情形仍然不容乐观。

作为高校学生，作为有志于为当今网络安全作出贡献的程序爱好者，我们以现有的入侵检测系统为原型，开发出一整套网络行为分析系统，并力争做到高效，易用，实用，健壮和强大。针对传统系统漏报误报率高，丢包率高，自身不够安全等情况，**AegisShield2.0** 基于多核平台的网络行为分析系统期望通过一系列措施改善这些缺陷，以提高系统的可信度和健壮性，在提供一套可靠的入侵检测功能同时还增加了内部用户网络行为的分析，包括用户使用的网络软件分析、浏览网站分析等。可以说，**AegisShield2.0** 把外部的行为和内部的行为分析有机地融合在一起，为管理员提供网络安全分析的有力工具。

形势是严峻的，我们明白自己的力量不足以改变网络安全现状，但是我们相信会有越来越多的人投身于信息安全的保障工作中。我们也会不断完善我们的系统，希望**AegisShield2.0** 能给大家带来新的福音！

参考文献

- [1] Snort manual
- [2] Ntzc Project, <http://code.google.com/p/ntzc>
- [3] Linux L7 Filter, <http://sourceforge.net/projects/l7-filter/>
- [4] 《Linux 设备驱动开发详解》, 人民邮电出版社, 宋宝华, 2008/2
- [5] 利用 OpenMP 线程绑定技术提升多核平台应用性能, 徐磊等
- [6] Linux Cross Reference, <http://lxr.oss.org.cn>
- [7] Linux 2.6 调度系统分析, IBM developerWorks, 杨沙洲
- [8] 计算机网络安全与防火墙技术[J]. 信息技术与信息化, 朱玉锦等. 2007.40-42
- [9] 基于防火墙与入侵检测联动技术的系统设计[J]. 武汉理工大学学报, 杨琼等 2005-7.
- [10] 分布式入侵检测系统的研究与设计[D]. 重庆大学硕士学位论文, 阳波. 2007-4. 6-56.
- [11] Sredom 谈谈 IDS 的检测与规则[J] www.xfocus.org 2003-3-16
- [12] 张丙凡, 李永忠, 范智勇 多元化入侵检测技术[J] 计算机仿真 2009-11
- [13] 周凯, 夏幼明 基于 Agent 技术和免疫机制的分布式入侵检测模型初探[J] 云南大学学报 2009, 31
- [14] 黄勤 基于 PCA 的 GABP 神经网络入侵检测方法[J] 计算机应用研究 2009-12
- [15] 顾明 基于粗糙集的自适应入侵检测算法[J] 清华大学学报 2008, 48
- [16] 王晓霞 基于神经网络模型分割的入侵检测方法[J] 计算机工程与设计 2009, 30
- [17] 洪伟 基于数据挖掘的数据库入侵检测系统的研究[J] 湖南第一师范学院学报 2009, 5
- [18] 曲进 基于无线局域网的入侵检测原理分析与实现[J] 信息与电脑 2009-11
- [19] 李继洪 基于用户行为统计的入侵检测判据研究[J] 微计算机信息 2009, 25
- [20] 崔炜 入侵检测系统的分析与应用[M] 电脑学习 2009-12
- [21] 徐仙伟 入侵检测系统中两种异常检测方法分析[J] 网络安全 2009-1