|  |  |
| --- | --- |
|  | 学校代码： 10246 |
|  | 学 号： 13210240095 |
|  |  |



|  |
| --- |
| 硕 士 学 位 论 文 |

（专业学位）

|  |
| --- |
| **一种虚拟化环境下针对分布式软件定义存储的性能监测与健康预警系统** |

|  |
| --- |
| A Performance monitoring and Health warning system for Distributed Software Defined Storage in Virtualization Environment |

|  |  |
| --- | --- |
| 院 系： | 计算机科学技术学院 |
| 学位类别（领域）： | 计算机技术 |
| 姓 名： | 刘晓通 |
| 指 导 教 师： | 吴承荣 副教授 |
| 完 成 日 期： | 2015年9月20日 |

**指导小组成员名单**

**张世永 教 授**

**钟亦平 教 授**

**吴承荣 副教授**

**吴 杰 研究员**

**吕智慧 副教授**

# 目录

[目录 1](#_Toc400470158)

[摘要 1](#_Toc400470159)

[Abstract 2](#_Toc400470160)

[第一章 绪论 1](#_Toc400470161)

[1.1研究背景 1](#_Toc400470162)

[1.2相关研究 3](#_Toc400470163)

[1.3本文的主要工作 3](#_Toc400470164)

[第二章 相关技术介绍 5](#_Toc400470165)

[2.1基于OpenStack的虚拟网络技术 5](#_Toc400470166)

[2.1.1 OpenStack网络总览 5](#_Toc400470167)

[2.1.2服务进程的物理机分布 6](#_Toc400470168)

[2.1.3物理机的网络连接 7](#_Toc400470169)

[2.1.4租户网络和基础网络 7](#_Toc400470170)

[2.2入侵检测系统snort 8](#_Toc400470171)

[2.2.1工作模式 9](#_Toc400470172)

[2.2.2工作流程 9](#_Toc400470173)

[2.2.3规则和插件 10](#_Toc400470174)

[2.3数据包抓取库libpcap和包过滤技术BPF 11](#_Toc400470175)

[2.3.1数据包抓取库libpcap 11](#_Toc400470176)

[2.3.2包过滤技术BPF 11](#_Toc400470177)

[第三章 虚拟化环境下的数据包抓取点 13](#_Toc400470178)

[3.1数据包抓取的两种方式 13](#_Toc400470179)

[3.2主机上的虚拟网络 14](#_Toc400470180)

[3.2.1典型的虚拟网络配置 14](#_Toc400470181)

[3.2.2计算节点上的网络配置 15](#_Toc400470182)

[3.2.3网络节点的网络结构 16](#_Toc400470183)

[3.3数据包抓取的需求 16](#_Toc400470184)

[3.4数据包抓取点的组合方案 18](#_Toc400470185)

[3.4实验验证 18](#_Toc400470186)

[3.4.1实验方法 18](#_Toc400470187)

[3.4.2实验结果 19](#_Toc400470188)

[第四章 数据包的抓取 20](#_Toc400470189)

[4.1 snort方案 20](#_Toc400470190)

[4.1.1实验方法 21](#_Toc400470191)

[4.1.2实验操作 21](#_Toc400470192)

[4.1.3实验结果 22](#_Toc400470193)

[4.1.4可能的改进 22](#_Toc400470194)

[4.2 libpcap方案 22](#_Toc400470195)

[4.2.1性能特性 23](#_Toc400470196)

[4.2.2 包过滤性能实验的步骤 24](#_Toc400470197)

[4.2.3包过滤性能实验的结果 24](#_Toc400470198)

[4.3多进程libpcap方案 25](#_Toc400470199)

[4.3.1架构的选择 25](#_Toc400470200)

[4.3.2包过滤规则的组合方式 27](#_Toc400470201)

[第五章 数据包信息合并 32](#_Toc400470202)

[5.1算法的要求与思路 32](#_Toc400470203)

[5.2算法描述 34](#_Toc400470204)

[5.2.1数据结构 34](#_Toc400470205)

[5.2.2算法操作流程 34](#_Toc400470206)

[第六章 系统详细设计与实现 37](#_Toc400470207)

[6.1系统功能和架构 37](#_Toc400470208)

[6.2数据包采集记录程序 38](#_Toc400470209)

[6.2.1初始化模块 38](#_Toc400470210)

[6.2.2协议解析模块 39](#_Toc400470211)

[6.3转存程序和数据存储中心 40](#_Toc400470212)

[6.4守护进程和控制中心 40](#_Toc400470213)

[第七章 系统实验 42](#_Toc400470214)

[7.1实验环境 42](#_Toc400470215)

[7.2功能验证 42](#_Toc400470216)

[7.3性能测试 43](#_Toc400470217)

[7.4数据包信息合并测试 43](#_Toc400470218)

[第八章 总结与展望 45](#_Toc400470219)

[参考文献 46](#_Toc400470220)

[后记 48](#_Toc400470221)

# 摘要

随着云计算技术的快速发展，越来越多的数据中心采用虚拟化网络作为基础网络架构。而数据包采集和分析作为最基础的网络技术，也就需要做出相应的调整，以适应虚拟化网络环境。同时，数据包采集的效率也应当达到一定要求。

本文对虚拟化环境下的数据包采集和记录问题进行了一系列研究，最终提出了一个虚拟化环境下的分布式高效数据包信息采集与记录系统。这个系统可以看做一个独立的信息采集系统，也可以当做安全审计的基础子系统来使用。这为进一步实现虚拟化环境下的安全审计系统提供了基础。

本文对虚拟化环境下的网络架构进行了研究，提出了这种环境下的有效的数据包抓取点组合方案。对几种可能的数据包抓取方案进行了探索，最终选择了基于libpcap的方案，并提出了一种基于BPF数据包过滤规则的多进程libpcap数据包采集方案。另外，本文针对采集的数据重复率高的特点，提出了一种基于哈希表、堆排序和估值函数的数据包信息合并算法，有效地压缩了所需的存储容量，缓解了数据量与存储容量不匹配的问题。

最终，本文设计并实现了目标系统，并对它进行了一系列的测试和验证。

**关键字：**虚拟化环境、多进程、数据包抓取、libpcap、信息合并

# Abstract

Along with the fast development of cloud computing technology, more and more data centers adopt virtual networking as their basic network architecture. As the most basic networking technology, packet collecting and analyzing need some adjustment according to virtual networking environment. And the efficiency of packet capture should reach some standard.

This article studies a series of problems in packet collecting and recording in a virtual networking environment, and proposes a distributed efficient packet information collecting and logging system in virtualization environment. This system can be treated as an independent information collecting system, or can be used as a basic subsystem of security audit system. These works lays the foundation of a security audit system in virtualization environment.

This article studies the networking architecture of virtualization environment, proposes an effective merging of packet capture point in this environment. It explores a number of packet capture plans, finally choses the libpcap plan, and proposes a multi-process packet capture plan based on Berkeley Packet Filter. Aiming at the high repetition rate, we propose a packet information merging algorithm which is based on hash table, heap sort and valuation function. This algorithm effectively compresses the storage space needed by the system, thus relieves the mismatch of data amount and storage volume.

As an end, this article designs and implements the target system, and applied a series of testing and verifications on it.

**Keywords:** Virtualization Environment, Multi Process, Packet Capture, libpcap, Information merging.

# 绪论

## 1.1研究背景

1.1.1 vsphere与云计算

1.1.2存储与软件定义存储(vsan)

1.1.3 vsan目前没有实现的功能

当前云计算的热潮方兴未艾，大多数有实力的互联网公司都已经或者正在建设自己的云计算平台。从亚马逊的EC2和阿里云，到百度云盘，这些新平台无一不在改变着人们使用计算机的方式。想拥有自己的网站？不用再自己维护服务器或者用传统方式租用价格高昂的服务器，只需要购入一台云主机。要在寝室和实验室的电脑之间共享文件，不用再用文件传输协议（FTP）或者远程桌面连接，只需要放到云端，就算出差时用别人的电脑也能轻松访问。



图1-1 云计算

云计算技术的快速发展也给数据中心的网络架构带来了深刻的技术变革。新的数据中心需要使用虚拟化的网络架构，来实现灵活性、开放性、高效性和易于管理等关键特性，以达到云计算技术的要求。在这样的网络中，可能广泛采用软件定义网络（SDN）、虚拟本地网（VLAN）、网络地址转换（NAT）和虚拟网桥等技术。新的数据中心网络架构带来了很多新的挑战，比如虚拟网络的流量监控、安全性和故障排除等。这就使得云计算数据中心需要能够适应网络虚拟化的数据包采集和分析系统。

数据包的采集分析器是能够截获并记录数字网络的所有或部分流量的一个计算机程序或计算机硬件。当数据流在网络上传输时，这个系统抓取每一个数据包，根据需要来解析原始数据，提取出数据包中各个字段的值，并进行分析。这些操作需要相应的RFC或者其他标准化文档作为依据。

数据包采集分析系统往往是网络管理工作必备的工具。分析网络故障、检测网络入侵、监控网络状态、调试网络协议栈和网络应用程序等工作，如果没有数据包采集系统都无法顺利进行。

例如，当我们进行网络编程时，往往需要调用操作系统的相关应用程序接口（API）。这些API的底层是操作系统实现的一套功能完备的网络协议栈。其中负责接收网络流量并解析、分发给用户程序的部分也可以看做是一个数据包采集分析系统，只不过它是系统内核的一部分，所以比较特殊。

例如，一个网络管理员需要对网络故障进行比较精细的分析时，或者测试网络浏览器的时候，往往会使用Wireshark之类的数据包分析软件对网络流量中一些关键的包进行仔细分析，才能在网络协议的一些字段中找到足够的信息。

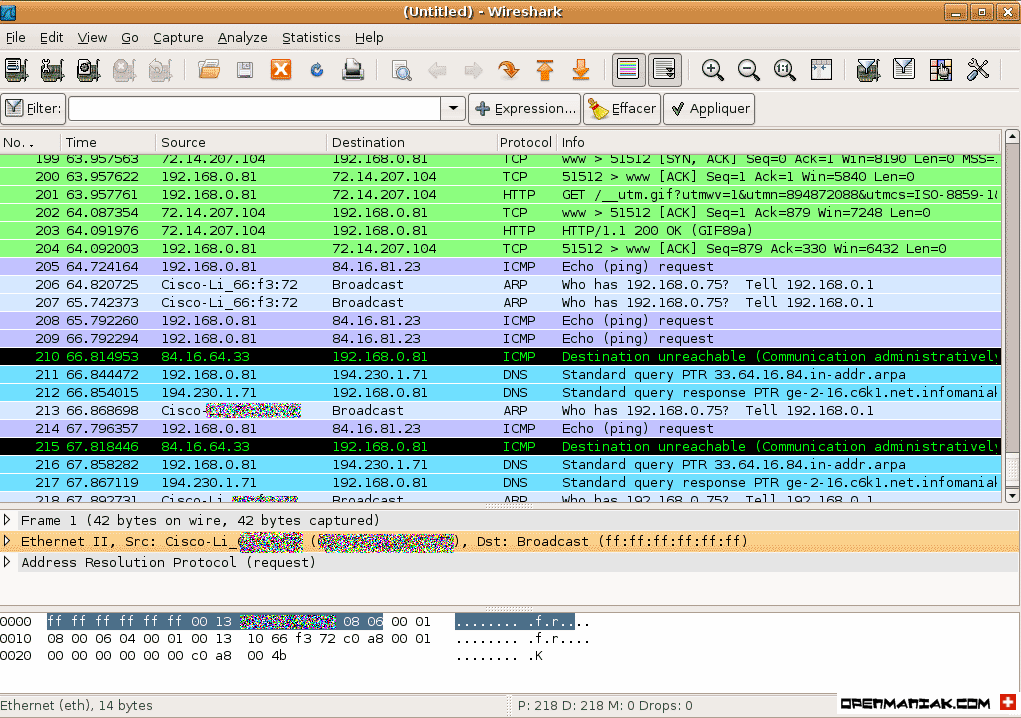


图1-2 Wireshark界面

除此之外，数据包的采集和分析还有另外一个重要的用途，那就是安全审计。

安全审计是指，对计算机系统运行过程中产生的信息进行实时的过滤和筛选，并将其中可能与安全风险有关的部分记录下来，并且在固定的时间或者在发生入侵时，对这些记录进行分析，寻找系统安全漏洞或黑客入侵手法。需要审计的信息包括执行的程序、访问的文件、发送和接收的网络流量等，其中网络流量的审计是安全审计的重要内容。

安全审计与上面提到的其它使用场景的一个最大的不同之处在于，安全审计系统需要对相关信息进行持续的记录。只要系统在运行，就需要把相关的信息存储下来。而这个特点导致了一个问题，那就是，受到存储能力的限制，我们往往无法把所有可能有助于安全审计的信息都记录下来，而只能对它们进行筛选，只记录其中关键的一小部分。如何有效地抽取有用的信息，抛弃冗余的信息，也是安全审计面临的课题之一。

在网络技术快速发展，云计算带来了网络虚拟化的当下，我们十分需要能够适应网络虚拟化的数据包信息采集器。同时，数据包信息的有效过滤和存储也是一个很现实的问题。所以，设计并实现虚拟化环境下的高效数据包信息采集记录系统是一个有着很强科研价值和实用价值的课题。

## 1.2相关研究

在与云计算、虚拟网络、数据包抓取等相关领域，国内外学者展开了广泛的研究，其中一些为本文的研究提供了参考和启发。

在论文[1]中，Francesco Fusco和Luca Deri指出了在网络监控应用运行在多核系统时可能出现的一些常见问题，并提出了相应的解决方案。他们还描述了一种可靠的支持多核的数据包抓取内核模块，可以用来实现高速数据包抓取。

Spring J.在论文[2]中指出，云计算网络模型有是七层模型，这些层次可以由云计算提供商或云计算用户来控制。他列举了用于增强云计算安全的一系列限制手段和审计方法。

Wen Shuguang和Xie Gaogang在libpcap的基础上设计并实现了一个支持多线程的数据包抓取库——libpcap-MT。libpcap-MT可以在内核模式中使用多个缓冲池高效地抓取和分发数据包，减少了线程同步和内存拷贝带来的性能影响。非锁定模式的多个缓冲池队列允许内核和线程并行地读写数据包[3]。

Giorgos Vasiliadis和Spiros Antonatos等人在他们的报告[4]中提出了把入侵检测系统snort与GPU结合，开发出了称为Gnort的基于显卡的高速网络入侵检测系统。

## 1.3本文的主要工作

本文的大部分研究内容和系统的主要功能都来源于中国银联研究院云平台安全审计系统项目的实际需求。本文最终实现的系统在该项目原型系统的基础上进行了很多改进。

本文的目标是找到一种高效的方法来抓取虚拟化环境下的数据包，并且对数据包中的信息进行提取和筛选，从而设计和实现一套分布式高效数据包信息采集与记录系统。

研究方式是，先探究虚拟化环境下的网络架构，找到此环境下的有效的抓包方法，在此基础上尝试几种可能的数据包抓取方案，对它们的有效性、高效性和灵活性进行实验测试，从而选取一种比较理想的抓包方案。然后研究如何有效地过滤数据包信息中的无用信息和冗余信息，把存储的信息量降到可接受的范围内。最后，基于上述研究设计并实现目标系统，并对该系统进行测试和试运行。

目标系统应当能够实现以下功能：

1. 能够高效地采集主机、网络设备上的网络流量，提取时间戳、源/目的IP地址、源/目的端口和TCP/UDP，并进行记录；
2. 采集数据包时能够按照设定的规则进行数据包的过滤；
3. 能够在记录数据之前对重复数据进行合并，以减少存储空间的开销；
4. 能够分布式地部署，覆盖数据中心指定的区域；
5. 数据被记录下来之后能够集中起来存储；
6. 数据包采集程序能够统一控制和管理，数据包的采集规则能够统一下发。

文章的结构安排如下：

第一章是绪论，主要介绍了云计算和数据包抓取等研究背景，说明了所选课题的研究价值。然后介绍了国内外的相关研究进展。最后提出研究方法和目标，以及本文的组织方式。

第二章介绍了与课题有关的一些技术，包括虚拟网络技术、入侵检测系统snort、数据包抓取库libpcap和数据包过滤机制bpf等。

第三章对如何在虚拟化环境下有效地抓取数据包进行了探索，找到了一种系统的、有效的数据包抓取点的组合方式，并且对有效性进行了实验验证。

第四章通过对几种数据包抓取方案的尝试，最终找到了适合主机和网络设备的数据包抓取方案。

第五章探讨了如何合并大量出现的数据包信息条目，提出了一种优秀的条目合并算法，并对算法进行了实验验证。

第六章结合前面几章对系统实现方式的探索，对整个系统进行了详细设计。

第七章对系统进行了进一步实验和初步的试运行。

第八章是对本文的总结和未来工作的展望。

# 相关技术介绍

**2.1基于OpenStack的虚拟网络技术**

OpenStack是当前的主流云计算平台之一，它采用的虚拟网络技术具有很强的代表性。同时，本文的研究也都是在OpenStack环境下进行。所以这里有必要先介绍一下OpenStack虚拟网络的一些技术细节。



图2-1 OpenStack

OpenStack是一个自由、开源的云计算软件平台。大部分情况下它被用作基础设施即服务的解决方案。它由一系列互关的项目组成。主要功能是控制数据中心的计算资源池、存储资源池和网络资源池。用户可以通过基于网页的展示平台（web-based dashboard）、命令行工具和应用程序接口（API）进行对系统进行操作。OpenStack的版权归属适用Apache License。

**2.1.1 OpenStack网络总览**

网络组件是OpenStack模块化体系结构的一个独立的组件。在这个系统中还有计算组件、镜像组件、身份认证组件和网络展示台等其它组件。网络组件和其它OpenStack组件都独立地运行，相互之间进行必要的交互。和这些组件一样，部署网络组件需要在多台主机上安装数个服务。

网络服务器使用neutron-server守护进程来提供开放的网络API，以便用户管理网络插件。一般来说，网络插件需要使用数据库。

如果部署方案中有专门运行集中计算组件的控制器主机，那么网络组件服务可以部署在那一台主机上。当然，网络组件是完全独立的，所以也可以部署在一台单独的主机上。根据配置的不同，网络组件可以包括以下部分：

* 插件组件： 运行于每一台物理机上，用于实现虚拟网络。
* dhcp组件： 为租户网络提供DHCP服务。
* l3组件： 在租户网络中提供L3/NAT转发服务，用来接入外部网络。
* 监控组件： 为租户网络提供L3流量监控。

这些组件通过RPC或API与neutron主进程通信。另外，网络组件通过以下几个途径与其它OpenStack组件通信：

* 网络组件调用身份验证组件来进行API请求的授权
* 计算组件通过标准API与网络组件通信。在创建虚拟机时，计算组件通过调用网络组件API把虚拟机的虚拟网卡接入特定的网络
* 网页展示台中集成了网络组件的API，使得管理员和租户能够通过网页来创建和管理网络服务

**2.1.2服务进程的物理机分布**

和其它OpenStack服务一样，网络组件可以分布在多台物理机上运行。它也可以将所有服务进程运行于同一台物理机上。或者，也可以把每个服务放在一台物理机上，然后对某些服务主机进行冗余备份。

在标准体系结构模板中，系统由一台云控制主机、一台网络主机和几台承载虚拟机的物理机组成。云控制主机和网络主机可以是同一台机器。但是，如果虚拟机需要相当大的网络带宽，一个独立的网络主机能够避免CPU带来的吞吐量瓶颈。

**2.1.3物理机的网络连接**

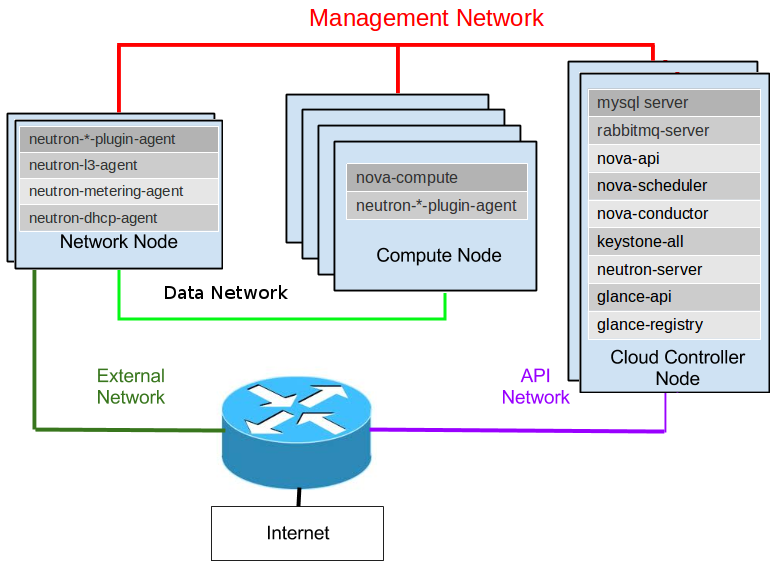


图2-2物理机的网络连接

网络组件的标准部署包括以下物理网络：

* 管理网：OpenStack组件之间的内部通信网络。这个网络中的地址应当只能够从数据中心内部访问。
* 数据网：云平台内部的虚拟机之间的通信网络。这个网络中的地址范围取决于采用了哪一种网络插件。
* 外部网：为虚拟机提供互联网连接。只要接入互联网就能够访问这个网络。
* API网：租户调用OpenStack API使用的网络。接入互联网的人都应该能够访问这个网络。API网和外部网可以是同一个网络，因为API网可以和外部网使用相同的IP地址段。

**2.1.4租户网络和基础网络**

下图展示了租户网络和基础网络的互联方式，并演示了它们在整个网络拓扑中的交互关系：

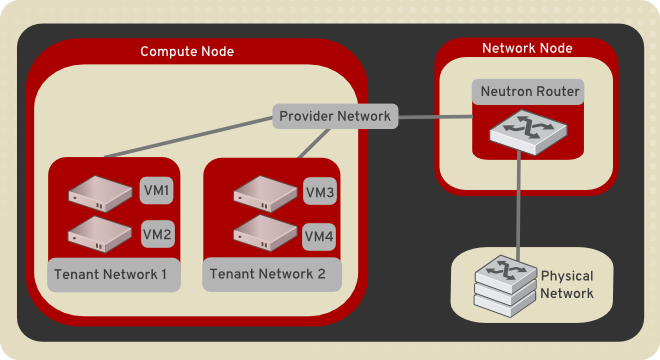


图2-3租户网络和基础网络

租户网络是指用户创建的、用户项目内部使用的网络。这种网络被默认设置为与外部完全隔离，是用户专用的独享网络。网络组件支持多种租户网络类型：

* 扁平网络：所有虚拟机都接入同一个网络。物理机也可以接入这个网络。不使用VLAN tagging或其他网络隔离技术。
* 本地网络：虚拟机只连接到所在的物理机，与外部网络隔离。
* VLAN：网络组件允许用户使用VLAN ID创建多个基础网络或用户网络，这些网络对应着物理网络中的不同VLAN。这使得虚拟机可以在系统范围内互相通信。
* VXLAN和GRE：VXLAN和GRE使用网络协议堆叠来提供虚拟机之间的直接通信。VXLAN通过将原来的VLAN包封装在UDP包中的方法提供了足够多的虚拟本地网编号。GRE通过创建隧道来保证虚拟机通信的独享性。

基础网络是指管理员创建的直接对应着数据中心的现有物理网络的虚拟网络。它的类型包括扁平网络和VLAN。基础网络可以共享给租户。

**2.2入侵检测系统snort**

基于snort的数据包抓取是本文尝试的多种抓取方案之一，所以在这里对snort的功能和特点进行一些介绍。



图2-4 Snort

Snort是一个开源的网络入侵检测系统，能够在多个平台下对IP网络进行实时的网络流量分析和包记录。它能够进行协议分析、内容搜索匹配，也能进行多样化的入侵检测，比如缓冲区溢出、隐藏端口扫描、CGI攻击和OS指纹提取等。

**2.2.1工作模式**

Snort很容易上手，但是要掌握所有命令行选项，并选取合适的组合选项并不那么容易。不过作为基础选项的工作模式只有三种，所有组合选项都属于这三种工作模式之一。这三种工作模式分别是：

* 嗅探器模式：

只是简单地从网络中读取报文，并且把它们连续地显示在屏幕上

* 记录器模式：

把报文记录到磁盘上的日志中

* 网络入侵检测系统（NIDS）模式：

对网络流量进行检测和分析。这是最复杂和可配置的模式。

### 2.2.2工作流程

Snort对数据包的处理大致分为五个步骤，每个步骤对应一个模块。这五个步骤分别是：

* 数据包抓取： 通过DAQ调用底层的相关函数库，来抓取数据包
* 协议解析： 解析数据包中的各层协议，提取协议字段
* 预处理： 重组IP包分片和TCP流、规范HTTP流、进行底层异常检测
* 规则匹配：将数据包与设计的规则进行匹配，来确定对包的处理方式
* 输出： 将处理信息输出到指定的目标，即屏幕或文件

### 2.2.3规则和插件

Snort具有极大的灵活性，这主要体现在用户可以自行配置检测规则，并且可以自己编写预处理插件和输出插件。

#### 2.2.3.1规则

Snort的规则文件由很多行规则组成，每行规则包括头部和选项两部分。例如：

\begin{figure}\begin{verbatim}alert tcp !192.168.1.0/24 any -> 192.168.1.0/24...
... 86 a5\vert''; msg:''external mountd access'';)\end{verbatim}
\par\end{figure}

图2-5 Snort规则样例

头部包括：

* 动作： 表示对匹配的数据包的处理方式，可以是alert（告警）、log（记录）或pass（忽略）等
* 协议： 匹配的协议类型，可以是IP、TCP、UDP、ICMP等
* 源/目的IP地址：可以使用子网掩码、并集和非运算等来表示一组IP地址
* 源/目的端口：可以给出上下限来表示一段连续的端口范围
* 传输方向：可以是传入、传出或双向

选项包含更强大的匹配方式，可以自定义，并且可以与插件配合使用。

#### 2.2.3.2插件

可以自定义的插件包括预处理插件和输出插件。用户编写好插件的代码之后，将其放入Snort源码中，重新编译安装，即可使用。用户编写的插件中可以调用Snort原有的各种数据结构和函数。

自定义的预处理插件可以用来进一步还原数据包，以应对攻击者通过新的手段来伪装攻击行为。

自定义的输出插件用来对数据包进行进一步的处理，以获取用户需要的特定信息，并且按照用户希望的格式和方式输出。

## 2.3数据包抓取库libpcap和包过滤技术BPF

Libpcap和Berkeley Packet Filter (BPF)是数据包采集方案中用到的底层数据包抓取和过滤机制，所以在这里进行简要的介绍。

### 2.3.1数据包抓取库libpcap

Libpcap是一个功能强大的原始数据包抓取库，供那些需要抓取网络流量的程序使用。Libpcap项目来源于Lawrence Berkeley Laboratory的Network Research Group的tcpdump项目。曾经属于tcpdump的底层数据包抓取、数据文件读取和数据文件写入的相关代码被抽取出来，组成了libpcap项目。

很多开源和商业网络工具都使用了libpcap提供的抓包和包过滤引擎，包括协议分析器、网络监控工具、网络入侵检测系统、流量生成器和网络测试工具。

### 2.3.2包过滤技术BPF

Berkeley Packet Filter (BPF)是类Unix系统中的一种底层包过滤技术。它提供了数据链路层的原始接口，使得发送和接收数据链路层的原始数据成为可能。如果网卡驱动支持混淆模式，它还可以把网卡设置为混淆模式，然后接收网络上的所有数据包。

更重要的是，它支持数据包过滤，使用BPF的程序只会接收到它们“感兴趣”的包。这可以避免把不需要的包从内核复制到用户程序，从而降低CPU和内存开销。BPF的过滤功能不是依靠过滤规则解释器实现，而是把规则编译成BPF虚拟机上的机器语言，从而极大地提高包过滤的效率。这种伪机器语言能够对数据包中的数据执行运算，然后根据运算结果决定是丢弃还是接收这个数据包。

其中最核心的技术就是，对于给定的匹配规则，如何转换成机器代码。我们可以使用tcpdump –d选项来查看对应的类汇编代码：

**root@NiSLPC:~# tcpdump -d 'ip and tcp'**

**(000) ldh [12]**

**(001) jeq #0x800 jt 2 jf 5**

**(002) ldb [23]**

**(003) jeq #0x6 jt 4 jf 5**

**(004) ret #65535**

**(005) ret #0**

这段代码的功能是：

* 载入偏移量为12的半字（half-word，2字节）
* 将这个值与0x0800比较，如果不相等则匹配失败。这一步检查了数据包是否是Ethernet上的IP包。
* 载入偏移量为23的字节。这是IP包的协议字段（protocol field）。
* 将这个值与0x6比较，不相等则匹配失败。0x6是TCP在IP头中的协议代码。
* 匹配成功，接收这个数据包。

# 虚拟化环境下的数据包抓取点

本章探讨在虚拟化环境下要从哪些位置或设备抓取数据包。这些位置或者说设备可以是主机和交换机的物理端口，也可以是虚拟网桥、网络命名空间等。我们首先介绍传统的数据包抓取方式，然后分析虚拟化环境的特点，最后探讨在这种环境下从哪些设备可以完整地抓取数据包。

## 3.1数据包抓取的两种方式

在传统的网络环境中，抓取数据包大致可以分为两种场景，本机抓取和流量镜像抓取。

本机抓取是指在目标主机上安装数据包抓取软件/硬件，直接对目标主机上的网络流量进行抓取。这种方式的特点是操作简单，无需做额外的准备工作，不需要额外的硬件。主机数据包抓取大都采用这种方式。

流量镜像抓取是指，在目标网络设备（路由器、交换机等）或目标主机上配置端口镜像，将数据流量复制一份，然后从专用的物理端口引出，用一台专用设备来处理镜像流量。这种抓取方式主要用于网络设备上的数据包抓取。在专用网络设备上，尤其是核心网络设备上，数据流量会很大。典型的数值是1Gbps到10Gbps。有些情况下甚至会更高。所以对流量镜像进行处理的设备需要有很高的性能。这些专用处理设备可能是高性能的服务器，也可能是专门为处理大流量开发的专用硬件设备。

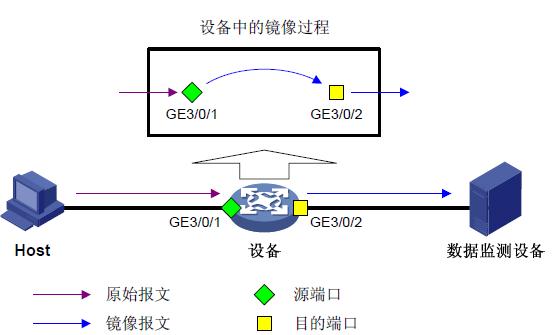


图3-1流量镜像拓扑图

在一般情况下，要监控整个网络，只需要监控所有主机流量或者监控所有网络设备即可。但是有时处于特殊需要，比如需要达到高等级的安全标准，也需要同时监控主机和网络设备。因为从网络设备上无法监控主机上发生的非法物理接触，而主机上无法监控到外部发起的某些针对网络设备的攻击。

在虚拟化环境下，数据包抓取也分为这两种方式。其中网络设备的流量镜像的抓取点与传统网络相同，只需直接抓取端口镜像流量即可。只是这些数据包可能比传统网络中多包裹了一些协议，用于实现虚拟网络。比如VLAN、GRE隧道等。这些是协议解析的内容，将在后面讨论。

而主机流量的抓取点就与传统网络大不相同了。在承载着虚拟机的主机上，部署着比较复杂的虚拟网络，用来把虚拟机连接到网络中来。虚拟机需要连接到它所在的物理主机、同一台物理主机上的其它虚拟机和外部网络。连接的方式也多种多样。所以我们主要讨论如何在承载着虚拟机的物理主机上选择合适的数据包抓取点。

## 3.2主机上的虚拟网络

在第二章，我们简要介绍了基于OpenStack的虚拟网络配置。这里，我们更详细地介绍虚拟网络在物理主机上的部署情况。这里以使用Open vSwitch插件的OpenStack虚拟网络为例。

### 3.2.1典型的虚拟网络配置

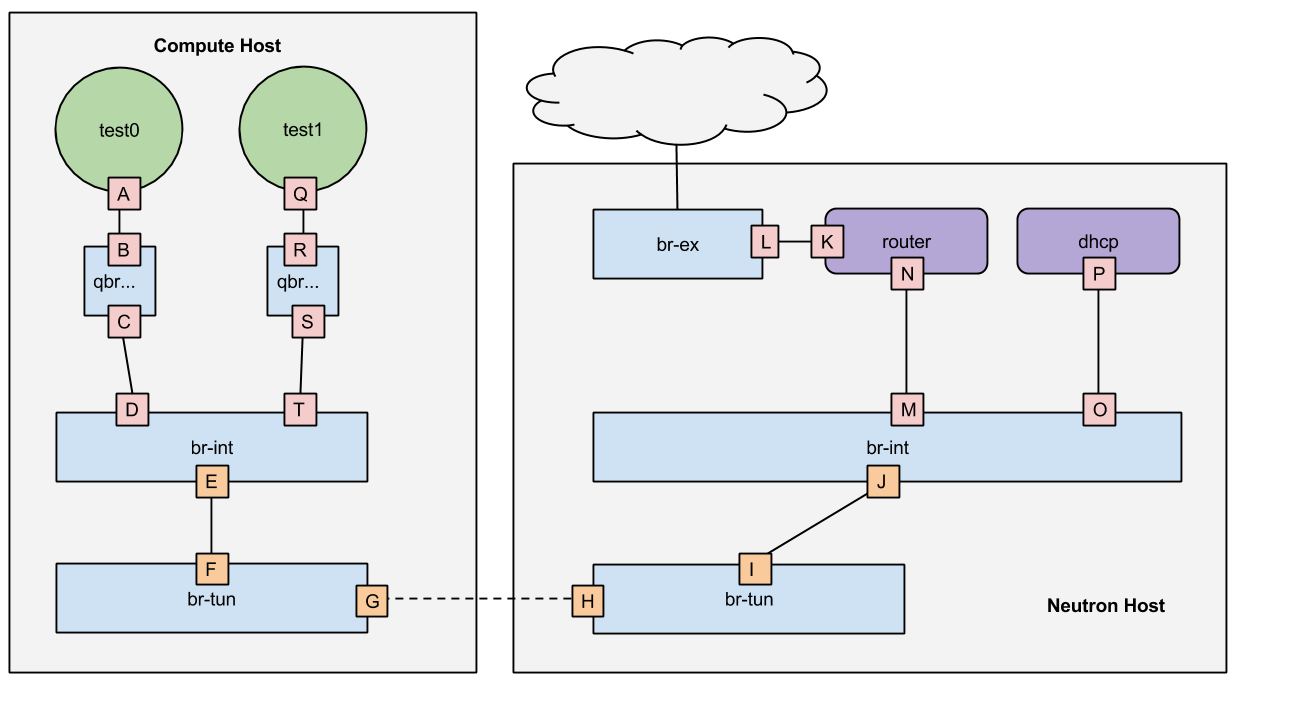


图3-2典型的虚拟网络连接

上图展示了一套典型的虚拟网络配置的细节。图中左侧方框中的是计算节点（Compute Host），用来承载虚拟机。右侧的是网络节点（Neutron Host），是整个虚拟网络的中枢，其它节点的网络流量都通过它来转发。虚拟机test0、test1接入计算节点，然后通过物理网络连接到网络节点，再接入到外部网络。

### 3.2.2计算节点上的网络配置

计算节点的主要功能是承载虚拟机，为虚拟机提供计算、存储和网络资源，并且向外部开放虚拟机的管理接口。在计算节点中，虚拟机通过一系列接口接入物理网络。计算节点内部的网络配置如下图所示：

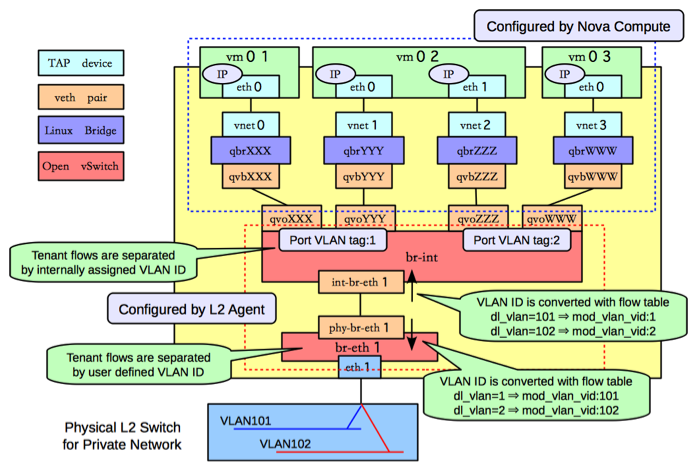


图3-3计算节点上的网络结构

在网络节点上有四种不同的虚拟网络设备：TAP、veth pair、Linux Bridge和Open vSwitch bridge。如上图所示，虚拟机vm01要通过eth0发送数据包到物理网络，需要经过九个设备。

TAP设备是虚拟机管理软件实现的虚拟网卡（vNIC）。送往TAP设备的数据包都由虚拟机操作系统接收。

Veth pair设备是一对直接连接的虚拟网络接口。从一个接口传入的数据包由另一个接口接收。它其实就是虚拟网线，用来连接其它虚拟网络设备。

Linux Bridge是虚拟网桥，它的作用类似于进行MAC转发的交换机。Linux Bridge可以连接物理设备和虚拟设备。它会根据数据包的目的MAC地址和转发表把接收到的数据包转发到相应的设备（端口）。

Open vSwitch bridge是虚拟交换机，它能够像物理交换机一样对端口进行配置，例如为一些端口配置VLAN。

在计算节点上，虚拟机的vNIC连接到命名为qbr的Linux Bridge，在这里实现OpenStack的虚拟网络防火墙，然后汇集到br-int，进行VLAN ID转换之后，连接到br-eth1，再连接到物理网卡eth1上，这样就到达了物理网络。虚拟机之间的通信在到达br-int之后就被转发到其它虚拟机，不会到达br-eth1。

### 3.2.3网络节点的网络结构

网络节点的主要功能是接收和转发各个节点之间、各节点与外部网络之间的通信，以及提供DHCP、虚拟路由器等内部服务。网络节点内部的网络配置如下图所示：

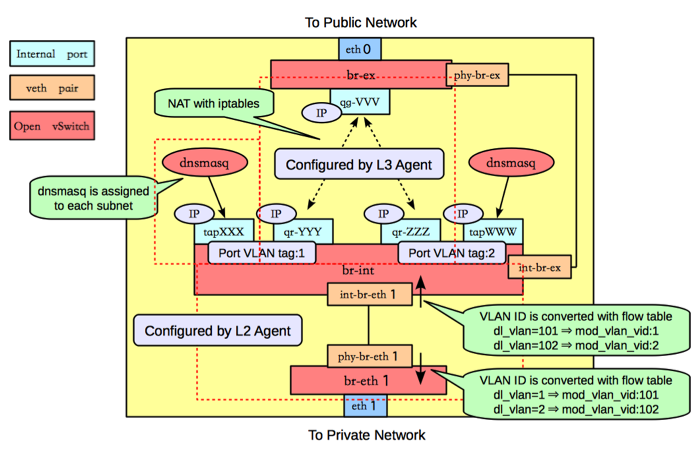


图3-3网络节点的详细网络配置

由于网络节点上没有虚拟机，不存在主机级的内部通信，所以这里不在对其网络配置进行详细分析。

## 3.3数据包抓取的需求

要选取合适的数据包抓取方案，首先明确数据包抓取的需求，也就是需要抓取哪些数据包。虚拟网络中的数据包可以根据来源和目的地的不同分为几种类型：

1. 主机内部通信

这里的主机可以是物理机也可以是虚拟机。主机内部通信是指同一个IP地址的两个不同的端口之间进行的通信。这类数据包由一个应用程序生成，交给操作系统的协议栈，由它转发给另一个应用程序。两次转发全部发生在软件层面，数据包根本没有在物理网络设备或虚拟网络设备上传输。抓取这类数据包需要调用相关的系统函数。

因为这类数据包是系统内部通信，不涉及网络安全或网络监控管理，所以大部分数据包分析软件都不抓取这类数据包。

1. 同一台物理机上虚拟机之间的通信

这类数据包的特点是，虽然有虚拟机的操作系统协议栈发出了，但是并没有送出物理机到达物理网络。它们只是经过qbr设备的防火墙之后，到达br-int，然后就根据VLAN ID和目的MAC地址转发给了另一台虚拟机。

这类数据包的存在是虚拟网络数据包抓取与传统网络中的主要不同之处。

1. 虚拟机与它所在的物理机之间的通信

一般来说，物理机的操作系统协议栈会只可能接入到专用的物理网卡或者桥接到br-int。这里的物理机指的是桥接到br-int上的物理机端口。数据包从vNIC经过qbr虚拟防火墙后，到达br-int，然后就被转发到物理机操作系统协议栈。

这类通信与上面一种类似，数据包在一台物理机上的虚拟设备上传输，没有进入物理网络。

1. 其它类型的主机间通信（经过物理网络的通信）

这类通信包括物理机之间的通信、虚拟机与其它物理机之间的通信、不同物理机上虚拟机之间的通信。vNIC和接入到br-int的物理机操作系统协议栈通过br-int和br-eth1连接到物理网卡。接入到物理网卡的物理机操作系统协议栈直接连接到物理网络。

下表总结了不同类型的数据包在传输过程中经过的虚拟/物理网络设备：

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 协议栈 | vNIC | br-int | 物理网卡 | 交换机/路由器 |
| 主机内部 |  |  |  |  |  |
| 物理机内  虚拟机之间 |  |  |  |  |  |
| 虚拟机与  所在物理机 |  |  |  |  |  |
| 虚拟机与  其它物理机 |  |  |  |  |  |
| 虚拟机与  其它虚拟机 |  |  |  |  |  |
| 物理机之间 |  |  |  |  |  |

表1-1各种数据包传输时经过的设备

在这几种数据包中，我们需要抓取的是第2、3、4种数据包。

我们不关心同一个主机（物理机或虚拟机）内部的通信，因为它们既不能反映虚拟网络的运行状况，也不能为系统安全分析提供有用的信息。

但是对于各种外部通信，包括第2、3种不经过物理网络，只在虚拟网络上传输的数据包，我们全部都要抓取。同一台物理机上的两台虚拟机之间也有可能存在黑客攻击，所以抓取这两类数据包对于安全审计也十分重要。

## 3.4数据包抓取点的组合方案

在抓取数据包时，除了要满足数据包抓取的需求之外，还应当考虑到系统性能。尽量早地去除重复的数据包是提高系统效率的一个重要要求。这个要求涉及到抓取点的选择和抓包过滤配置等很多方面。为了满足这一要求，就需要尝试各种抓取点的组合方案，寻找一个能够同时满足完整性和高效性需求的方案。

在目标虚拟网络中，可能的数据包抓取点有以下几个：

1. 虚拟机上的网络设备：从虚拟机内部抓取虚拟机上的流量。需要在虚拟机上安装抓包软件。
2. br-int：抓取Hypervisor上的br-int虚拟网桥，能够抓到虚拟机和连到br-int的物理机端口的流量
3. 未连接到br-int的物理网卡
4. 交换机/路由器端口镜像

在上述四个抓取点中，第1种需要在每个虚拟机上安装抓包软件，系统改动量大。而且它可以被第2种抓取点替代。第2、3、4中抓取点都是必需的。

所以一种合适的抓取点组合方案是，抓取所有br-int、未连接到br-int的物理网卡和交换机/路由器端口镜像。

## 3.4实验验证

为了验证抓取点组合方案，我们在不同的主机上使用ping命令产生数据包，发往不同的主机，同时使用tcpdump命令在不同的位置抓包，来测试能否在这些位置抓到特定种类的数据包。

### 3.4.1实验方法

这里以物理机之间的通信为例，演示在物理网卡上抓取数据包的验证方法。

1. 从物理机A ping物理机B，它们的IP地址分别为10.10.90.128、10.10.90.171

**ping 10.10.90.171 > /dev/null &**

1. 在物理机A上监听eth0，寻找相关数据包

**tcpdump -i eth0 | grep 10.10.90.171**

物理机A上的输出如下图：

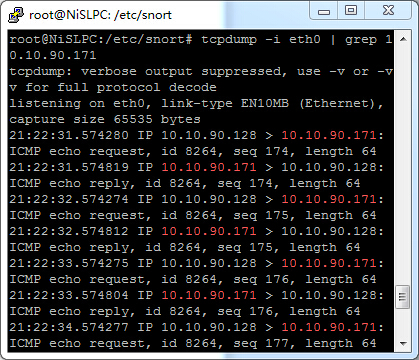


图3-4抓取点验证实验示例

### 3.4.2实验结果

抓取点组合方案验证实验结果如下表所示：

|  |  |  |  |
| --- | --- | --- | --- |
|  | br-int | 物理网卡 | 交换机/路由器 |
| 同一台物理机上的虚拟机 |  |  |  |
| 虚拟机到所在物理机 |  |  |  |
| 虚拟机到其它物理机 |  |  |  |
| 虚拟机到另一台物理机上的虚拟机 |  |  |  |
| 物理机到物理机 |  |  |  |

表3-2抓取点组合方案验证实验结果

上面的结果显示，同时抓取br-int、物理网卡和交换机/路由器镜像能够捕捉到所有类型的数据包。所以这种抓取点组合方案是有效的。

# 数据包的抓取

本章中，我们将尝试几种可能的数据包抓取方案，寻找其中最符合需求的，作为目标系统的数据包抓取方案。对抓包的需求主要是包抓取和包过滤的CPU开销小，包抓取的吞吐量达到一定要求。

## 4.1 snort方案

我们已经在第二章介绍了snort的基本信息。基于snort的数据包抓取方案中，我们使用snort的数据包抓取和协议解析功能来获取数据包中的信息，同时使用snort规则来实现包过滤，最后使用自行编写的输出插件把需要的字段按照需要的格式输出到文件中。

Snort支持的包过滤规则包括协议类型、源/目的IP、源/目的端口、数据包流向等基础规则，也包括分组偏移量等IP字段、TCP flags、HTTP字段等高级规则。

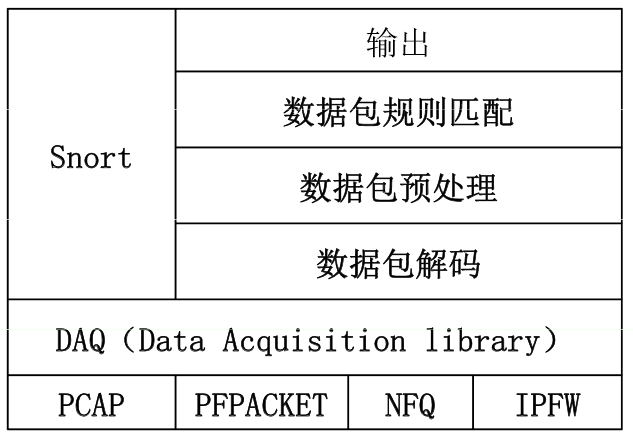


图4-1 snort内部的处理流程

我们首先实现了一套功能比较简单的测试用系统。在这个系统中，我们按照上述方案实现了数据包抓取、协议分析的功能，并且把协议字段直接输出到文件中。这个系统中不包含数据包去重功能。

接下来我们对这个系统进行了一系列功能和性能测试。这些测试表明系统的功能符合期望，但存在一个严重的性能问题，那就是它的包过滤效率没有达到预期。

Snort方案的问题涉及到三个实验，分别是全流量抓取性能实验、指定IP地址过滤性能实验和TCP flags过滤性能实验。这一小节就详细描述这三个实验。

这里使用的snort版本为Version 2.9.6.0 GRE (Build 47)。

### 4.1.1实验方法

这三个实验的方法相同，只是配置不同。每个实验都分为以下几个步骤：

1. 关闭所有可能产生明显网络流量的进程，确保实验开始前网络流量几乎为零
2. 配置测试用系统的包过滤规则
3. 选择合适的端口，启动测试用系统开始抓包
4. 开启CPU监控工具
5. 从内网高速下载一个大文件，期间记录CPU使用率，直到下载结束
6. 记录测试用系统使用的CPU时间

三个实验的snort包过滤规则分别设置为：

1. 抓取所有数据包

**log ip any any -> any any**

1. 抓取源/目的IP地址都不是下载服务器IP的数据包

**log ip !10.10.90.128 any -> !10.10.90.128 any**

1. 抓取TCP flags中含有tcp-syn的数据包

**log tcp any any -> any any (flags:+S; sid:10001;)**

### 4.1.2实验操作

下面以全流量抓取性能实验为例，演示测试操作流程。

1. 查看eth0端口十秒内的总流量变化，确定没有明显流量

**root@NiSLPC:~# ifconfig eth0 | grep bytes; sleep 10; ifconfig eth0 | grep bytes**

**RX bytes:2138687726 (2.1 GB) TX bytes:59703271 (59.7 MB)**

**RX bytes:2138695943 (2.1 GB) TX bytes:59705118 (59.7 MB)**

1. 改写snort配置文件

**root@NiSLPC:~# echo 'log ip any any -> any any' > /etc/snort/snort.conf**

1. 启动测试用系统

**root@NiSLPC:~# snort -i eth0 -c /etc/snort/snort.conf > /dev/null &**

**[1] 8359**

1. 开启CPU监控

**root@NiSLPC:~# top -p 8330**

**%CPU 0.0 TIME+ 0:00.02**

1. 下载文件

**root@NiSLPC:~# scp 10.10.90.128:/home/wx/file .**

1. 采集数据

### 4.1.3实验结果

预期的实验结果分别是：

1. 测试用系统CPU使用率明显不为零，CPU时间较长
2. 包过滤机制很高效地过滤了含有下载服务器IP的流量，CPU使用率几乎为零，CPU时间几乎为零
3. 包过滤机制很高效地过滤了不含tcp-syn的流量，CPU使用率几乎为零，CPU时间几乎为零

实验过程中下载速度维持在11MB/s左右。实验结果如下：

1. 平均CPU使用率7.0%，CPU时间总和0:06.86。
2. 平均CPU使用率11.3%，CPU时间总和0:09.74。
3. 平均CPU使用率8.7%，CPU时间总和0:06.84。

实验结果表明，snort方案无法满足包过滤效率的需求。我们必须寻找能够更高效地实现包过滤的方案。

### 4.1.4可能的改进

经过进一步调研发现，snort支持底层的bpf过滤机制，可以更高效地进行数据包的过滤。这种过滤机制是利用snort调用的底层网络库libpcap实现的。

如果包过滤使用bpf实现，那么snort的作用就只有协议解析了。而我们对协议解析的要求并不高，只需要解析出几个关键字段。所以还不如直接调用libpcap来实现数据包采集。

## 4.2 libpcap方案

在libpcap方案中，我们直接调用libpcap库来监听目的端口。同时，使用libpcap的bpf过滤机制，来实现数据包的过滤。网络流量经过bpf过滤，然后被libpcap抓取并传递给自行编写的协议解析模块。经过对Ethernet、VLAN、IP、TCP/UDP、HTTP、FTP等协议的解析，获取需要的字段，然后把数据输出到硬盘。

### 4.2.1性能特性

其中，用户定义的包过滤规则会被libpcap编译成bpf机器代码，传递给linux kernel的LSF（Linux Socket Filtering，BPF的linux实现），并运行在内核态。这样，包过滤的性能就能得到极大的提升。

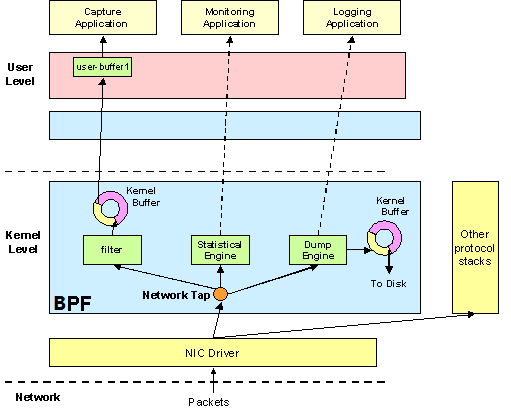


图4-2 libpcap应用程序内部架构

同时，在新版本的linux kernel和libpcap的组合下，可以开启零拷贝（zero-copy，也称为mmap）技术。在没有开启零拷贝的情况下，抓包过程效率很低，它使用很有限的缓存空间，而且没抓取一个包都需要进行一次系统调用。如果像libpcap那样要获取时间戳，还需要一次额外的系统调用。而zero-copy则高效许多。它提供一个可以映射到用户空间的大小可配置的缓冲环（circular buffer），用于接收和发送数据包。这样，一个系统调用就可以把多个数据包送到缓冲环中，而且应为内核和用户共享缓存空间，所以不需要再把数据从内核空间拷贝到用户空间，从而减少了开销。启用零拷贝需要kernel版本2.4/2.6/3.x和libpcap 1.x。

根据上面的方案，我们实现了测试用系统，并且进行了性能测试。

### 4.2.2 包过滤性能实验的步骤

Libpcap方案的测试方法与snort方案的相同，这里就不在重复。三个实验的包过滤配置如下：

1) 抓取所有IP数据包

ip

2) 抓取源/目的IP地址都不是下载服务器IP的数据包

not host 10.10.90.128

3) 抓取TCP flags中含有tcp-syn的数据包

(tcp[tcpflags] & tcp-syn) = tcp-syn

实验操作与snort方案类似，就不再赘述。

### 4.2.3包过滤性能实验的结果

实验的期望结果如下：

1. 测试用系统CPU使用率明显不为零，CPU时间较长
2. 包过滤机制很高效地过滤了含有下载服务器IP的流量，CPU使用率几乎为零，CPU时间几乎为零
3. 包过滤机制很高效地过滤了不含tcp-syn的流量，CPU使用率几乎为零，CPU时间几乎为零

实验过程中下载速度维持在11MB/s左右。实验结果如下：

1. 平均CPU使用率8.2%，CPU时间总和0:07.11。
2. 平均CPU使用率0.0%，CPU时间总和0:00.01。
3. 平均CPU使用率0.0%，CPU时间总和0:00.00。

实验结果符合预期。Libpcap测试用系统的数据包处理性能与snort测试用系统相近。对于IP地址和TCP flags的包过滤，libpcap测试用系统的CPU开销几乎可以忽略。这一点大大优于snort测试用系统。

## 4.3多进程libpcap方案

Libpcap方案已经较好地解决了主机上数据包抓取的问题，但是对于交换机/路由器的端口镜像，它的处理能力就显得不足了。核心交换机的端口镜像是把多个物理端口的流量镜像到一个万兆网卡，峰值流量能达到3Gbps。所以我们需要寻找一种专门用于处理端口镜像流量的数据包抓取方案。

使用多进程来处理单个CPU核心无法处理的任务是一种传统而有效的方法。这一小节就来探讨如何设计多进程的libpcap数据包抓取方案。

### 4.3.1架构的选择

要使用多个CPU来处理流量，最重要的问题是，如何把流量分发到多个进程或线程。

#### 4.3.1.1方案一

最容易想到的方法是，使用多线程，其中一个线程负责抓取数据包，并且把数据包均匀地分发给其他线程。其他线程则负责进行协议分析等数据包处理工作。数据包的处理过程如下图所示。其中抓包模块和三个协议解析模块都分别是一个进程。

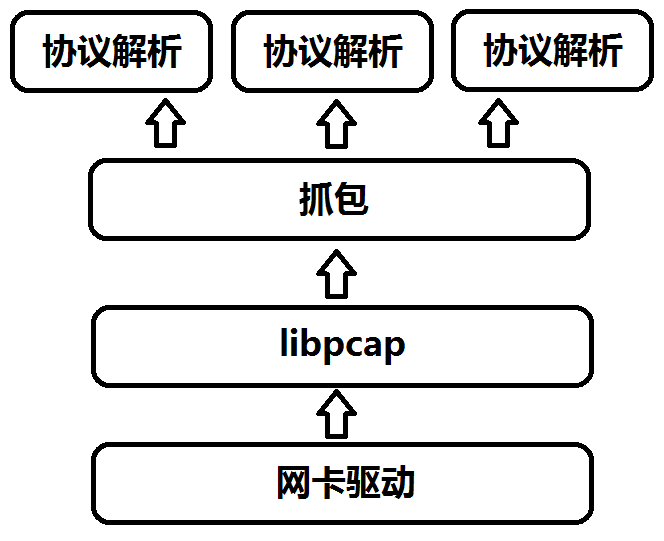


图4-3多线程/进程方案一

但是这种方案中，所有的流量都会流经负责分发的线程，这个线程可能成为瓶颈。事实上，在抓取数据包的过程中，CPU开销最大的不是协议分析和数据包的后续处理，而是把数据包传送到内存中的指定缓冲区的过程。所以，只把协议解析等数据包处理工作分散到多个CPU核心上是无法处理大流量的，数据包抓取的过程仍然会成为系统的瓶颈。

另外，libpcap的包传递机制也不支持这种数据包的并行处理。当负责抓包的线程把数据包传递给协议解析线程时，不可能把数据包拷贝一份，而是把内存指针传递给过去。这个指针实际上指向的是libpcap开辟的内核态与用户态共享的指针。Libpcap的缓冲区机制要求用户程序处理完一个数据包之后再去获取下一个数据包的指针。它不保证获取了新的数据包指针之后上一个指针仍然有效。

#### 4.3.1.2方案二

既然数据包抓取是CPU开销最大的步骤，那么就要考虑在抓取数据包时就开启多个进程或线程。

在这种方案中，直接使用多个相互独立的抓包进程，分别进行数据包过滤、数据包抓取和协议分析等进一步处理工作。数据包的分发则由bpf规则来实现。通过设置不同的bpf包过滤规则，使得不同的数据包分别与不同进程中设置的bpf包过滤规则相匹配，进而被不同的进程抓取和处理。

下图展示了多进程抓包方案的内部结构。其中，每一个进程都有一组独立的libpcap-抓包-协议解析模块。上图中，左、中、右三列分别对应三个进程，它们在内核中设置的bpf分别从网卡驱动获取数据包。

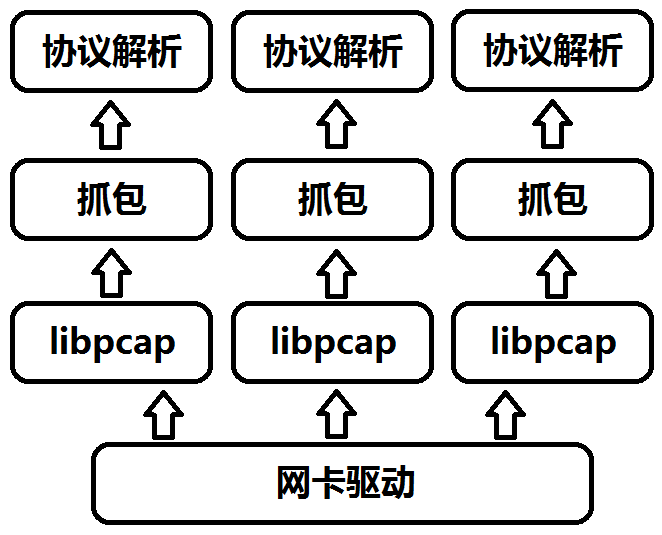


图4-4多线程/进程方案二

因为方案一中仍然有无法避免的瓶颈，而方案二的架构则比较合理，所以接下来的分析都是针对方案二进行的。

### 4.3.2包过滤规则的组合方式

在上面选取的架构中，采用一组包过滤规则来对网络流量进行分发。多进程架构中的每一个进程对应着一条包过滤规则。要把流量均匀地、无重复地分发到各个进程，就意味着对每一个数据包都有且只有一条规则与之匹配，并且与每条规则匹配的概率都相等。

#### 4.3.2.1负载均衡问题

我们把所有可能的数据包组成的集合记作A。把其中需要抓取的数据包的集合记作S。S对应着采用单进程的抓包方案时的包过滤规则R。把进程数量记作n。把多进程使用的包过滤规则R1、R2、R3…Rn对应的数据包集合分别记作S1、S2、S3…Sn。数据包a出现的概率记作P(a)。

则S1、S2、S3…Sn应当满足以下条件：

1. 对任意、，若，则
2. 
3. 对任意，

包过滤规则的匹配实际上是对数据包中的数据，或者说字段，的值进行匹配。把这些字段记作a1、a2、a3...，则P(a)可以表示为

所以，如果能用一个或多个概率分布比较均匀的字段把数据包空间A分割为n个对称的、等概率的子空间，然后与S取交集，就可以满足上述条件。这里说的对称是指空间A分割得到的n个子空间对S来说对称，也就是说这些子空间与S取交集之后得到的n个新的子空间的位置对称、体积相同、概率密度分布相同。

这就要求选取的字段（数据包中的数据）不但要概率分布均匀，而且不能出现在原始匹配规则R中。

另外，因为这些匹配规则要用来匹配所有我们需要的数据包，所以就必须使用所有数据包都含有的下层协议的字段，如Ethernet、IP、TCP、UDP。TCP和UDP流量可以分别使用多进程处理，所以它们的字段也可以使用。如果某种应用层协议的流量很大，也可以另外开启几个进程来专门处理，这时就可以使用这些应用层协议中的字段。当然，前提是bpf过滤规则能够支持基于这些高层协议字段的包过滤。

#### 4.3.2.2近似均匀分布的字段

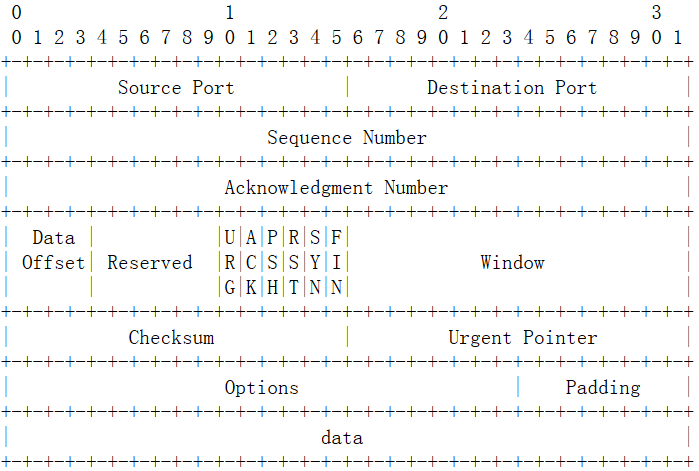


图4-5 IP报文的结构

考察了IP、TCP等协议内的所有字段之后，选出了可以用作分流因子的字段：

1. TCP序列号（TCP Sequence Number）

TCP序列号是TCP协议中用来标识数据在TCP流中的编号的字段。它的计算方式是，建立连接时生成一个初始序列号（Initial Sequence Number，ISN），然后每发送一个包，序列号就加上这个包里负载数据的长度。

因为序列号采用这种增长方式，所以在同一个TCP连接中，对序列号取模的结果是近似均匀分布的。

另外，为了应对基于IP伪装的TCP序列号猜测攻击（sequence number guessing attacks），当前版本的协议栈都使用随机性非常强的、难以预测的ISN。ISN也是近似均匀分布的。

所以，不论两个数据包是否属于同一个TCP连接，它们的序列号的模数都满足相互独立的近似均匀分布。

1. TCP应答号（TCP Acknowledgment Number）

TCP应答号表示数据包发送者希望接收的下一个序列号。所以这个字段的模数也满足近似均匀分布。

1. TCP校验和（TCP Checksum）

TCP校验和是由伪IP报头、TCP报头和负载数据进行两字节宽的反码求和得到的。这些数据包含了IP报头的部分字段、TCP报头的所有字段和所有负载数据，所以它的概率分布比TCP序列号更强。

1. IP报头校验和（IP Header Checksum）

计算方式与TCP校验和类似，校验的数据是整个IP报头。所以它综合了源/目的IP地址、IP报文长度、生存时间（Time to Live，TTL）等字段的分布特性。在交换机端口镜像的使用场景下，只要没有出现少数几对主机占用大部分带宽的情况，就能够保证这个字段的近似均匀分布。

1. UDP校验和（UDP Checksum）

计算方式和校验的内容与TCP校验和类似，校验的数据是伪IP报头、UDP报头和负载数据。由于对负载数据进行校验，所以满足近似均匀分布。

这几个字段按照概率分布的均匀性从高到低的排序序列是，TCP校验和、TCP序列号、TCP应答号、UDP校验和、IP报头校验和。

#### 4.3.2.3硬件加速校验和计算带来的问题

在上面列出的近似均匀分布的字段中，有三个字段时各个协议的校验和。但是在数据包抓取的过程中，并不总能够获取到正确的校验和。

校验和的计算并不总是由操作系统协议栈完成，也可能由网卡驱动甚至网卡硬件来完成。例如，以太网传输硬件会计算以太网CRC32校验和，而接收的硬件会检查这个校验和的正确性。如果校验和不正确，则会被直接丢弃。

高层协议的校验和则往往由协议栈计算，然后再把数据包交给硬件。但是版本较新的硬件现在也可以进行一些高层校验和的计算，如IP报头校验和、TCP校验和和UDP校验和。这种功能被称作checksum offloading。协议栈和网卡驱动不在自己计算校验和，而是直接把未经校验和计算的数据包交给硬件。

如下图所示，操作系统和网卡驱动层面都不进行校验和的计算，而是把它交给网卡硬件。在那里，同时进行基于MTU的数据报分组操作和各层校验和的计算。

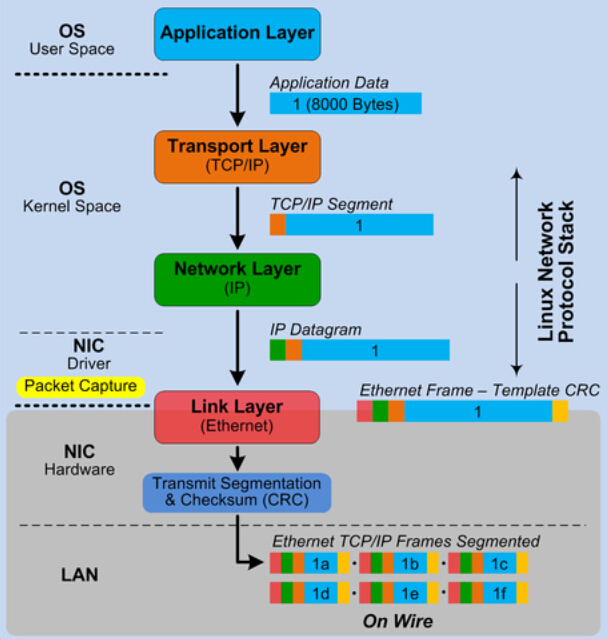


图4-6硬件加速的校验和计算

这就会导致libpcap抓取到的数据包是未经校验和计算的数据包，它们的校验和字段是全零或其它不确定的值。如果发生这种情况，这些字段的概率分布就不再满足近似均匀分布。这时我们就不能采用这些校验和字段来实现负载均衡。

这种情况发生的充要条件是，下面三个条件被同时满足：

1. 数据包的抓取点在发送这个数据包的主机上
2. 这台主机支持checksum offloading
3. 操作系统开启了checksum offloading功能

因为后两个条件与主机配置有关，属于不可控要素，所以要通过第一个条件来避免错误校验和的出现。

我们的多进程抓包方案是计划用于交换机端口镜像流量的处理的。这些数据包已经由网卡发出，在网络上传播，所以拥有正确地校验和。所以用于端口镜像的多进程抓包方案可以采用校验和字段来实现负载均衡。

而如果要把这套方案用于物理主机的抓包，那么基于校验和的负载均衡就只对入站流量有效。

#### 4.3.2.4分流字段的组合方式

因为我们设计多进程抓包方案是为了处理交换机端口镜像流量，所以就不必考虑未经计算的校验和的问题。我们在负载均衡的设计中将使用校验和，所以这里提出的分流字段的组合方式不能用于部署在物理主机上的系统。

综合考虑所有可以用于负载均衡的字段，因为IP层字段的概率分布均匀性较差，所以我们采用对TCP和UDP流量分别处理的方案。

对于TCP流量，因为TCP校验和的概率分布特性最好，而TCP序列号和TCP应答号也用于计算TCP校验和，所以我们采用TCP校验和而不采用TCP序列号和TCP应答号。用TCP校验和加上IP报头检验和，然后取模。

对于UDP流量，用UDP校验和加IP校验和，然后取模。

因为取模运算（^）要求linux kernel版本为3.7或更高，为了提高系统的适用范围，我们用按位与（&）来代替取模。

假如用来处理TCP流量的进程数量为，处理UDP的进程数量为，那么第个TCP进程使用的附加包过滤规则是：



第个UDP进程使用的附加包过滤规则是：

以八进程为例，假设原始过滤规则为original\_rule，则TCP和UDP的bpf过滤规则分别是：

(( ip[10] + ip[11] + tcp[16] + tcp[17] ) & 7 = i) and original\_rule

(( ip[10] + ip[11] + udp[6] + udp[7] ) & 7 = j) and original\_rule

将这种附加规则与人工设置的原始规则用“与”运算符链接就可以实现流量的多流量负载均衡。

# 数据包信息合并

我们在记录数据包信息时，只记录IP、端口、TCP/UDP和时间戳，所以在记录中会有大量重复的记录。为了减少记录的条数，降低系统所需的存储空间，提升查询效率，我们需要寻找一种算法来合并那些大量重复的记录。

## 5.1算法的要求与思路

要从实时生成的大量数据中找到并合并重复的条目，最好的方法是使用哈希表。哈希表的特点是，查询一条记录是否存在平均只需要常数时间。

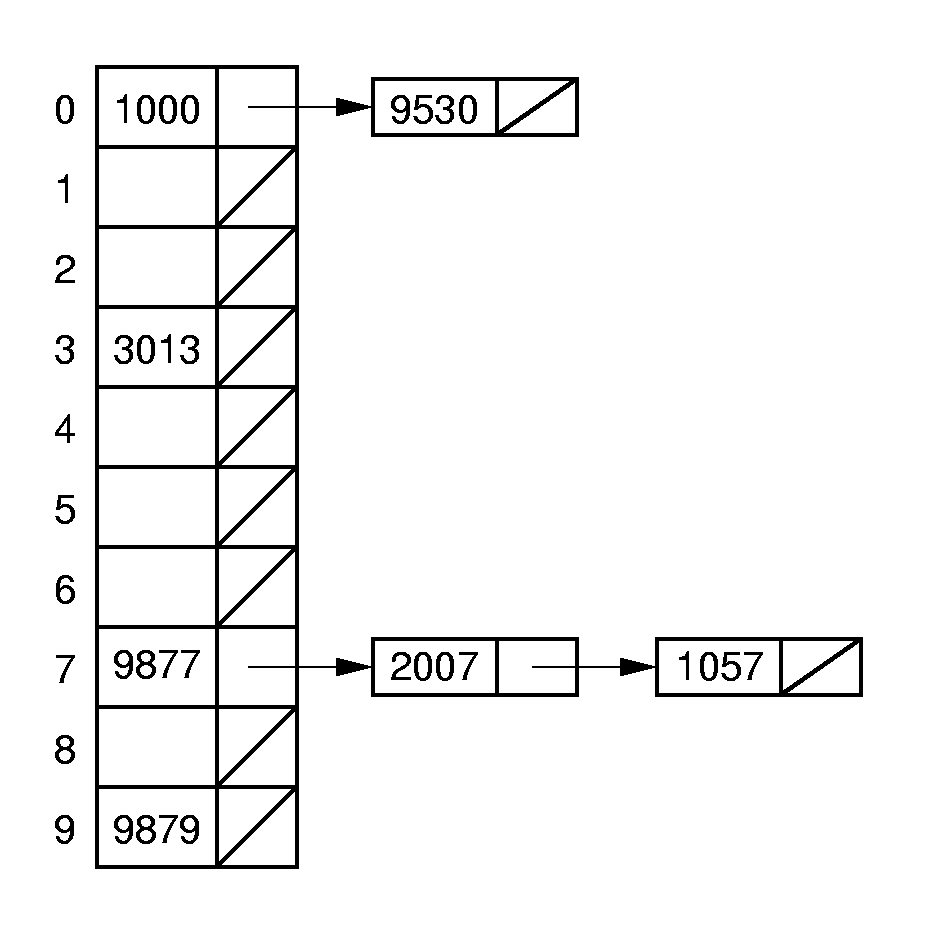


图5-1哈希表

但是，实际需求并不是这么简单。在实现快速地合并重复的条目的同时，还需要考虑一下这些问题：

1. 从哈希表中删除旧条目

哈希表有一个很重要的参数叫做装填因子，它的定义是已存储的条目的数量与地址空间的比值。使用哈希表进行高效查找的前提之一是装填因子被控制在一定范围内。当装填因子过大时，也就是哈希表中存储了太多条目时，平均查询时间会变得很长。

所以，为了保证算法在长时间连续运行时也能一直保持高效，就必须把一些不再需要的旧条目删除。

1. 优先删除重复率低的条目

既然要在运行的过程中动态地删除哈希表中的部分条目，那么为了达到条目合并的最大化，我们就需要优先删除那些看起来不容易发生合并的条目。也就是说，我们要寻找一种方法来高效地预估所有条目在接下来的时间段中出现的次数，然后优先删除那些出现可能性比较小的条目。

1. 在合理的时间内输出条目

条目的合并并不是终点，合并之后还需要选择合适的时间点把条目输出。对大部分条目来说，只要在它们从哈希表中删除的时候进行输出就可以了。

但是可能出现一种特殊情况，那就是，有一些记录总是重复出现，所以在一段非常长的时间里都没有被输出。直到某个时间，它的出现次数下降，然后之前的所有条目被合并为一条输出出来，并且这个条目的“出现次数”字段异常的大。为了应对这种特殊情况，就需要设定一个上限，当一个条目的存在时间或者出现次数超过一定数值时，即使这个条目的重复率仍然很高，也要把它输出出去。

面对上面列举的特性，我们很容易联想到一种基础的缓存调度算法——最久未使用算法（Least Recently Used，LRU）。如果把哈希表和链表实现的LRU组合起来使用，可以基本满足上述功能需求。

LRU的执行方式很简单，就是当需要删除一条记录的时候，找出最长时间没有被访问过的条目就行删除。这里可以使用双向链表来实现。哈希表中的每个条目对应着链表中的一个节点。这些节点按照上次被访问的时间排列。每次需要删除条目时，就删除链表尾部的节点以及哈希表中对应的条目。每次合并一个条目时，就把对应的链表节点取出，然后放到链表头部。

使用这样的组合数据结构，在设置合适的条目数上限以达到理想的装填因子的前提下，条目的合并操作可以在常数时间内完成（即渐进时间复杂度O(1)）。

但是，在某些特殊情况下，哈希表加LRU的算法效率会很低。其中一种情况就是，当有大量条目只出现一次或少数几次的时候。这时，这些重复次数很少的条目就会把出现次数很多的条目快速地挤到LRU队列的末尾，使得总合并次数大大减少。结合我们的应用场景，有很多网络流量都是同一对端口只有三五个数据包，例如HTTP访问等。

所以哈希表与LRU的组合并不是最合适的算法。经过多方面的考虑，我们最终选定了下面所述的算法。

## 5.2算法描述

算法的总体思路是，使用哈希表来记录和查找出现过的条目，另外建立一个堆，使用堆排序来找出哈希表中下一个要删除的条目，而进行排序的依据是一个估值函数对条目重复率的估计值。

### 5.2.1数据结构

这个算法使用的数据结构有一个哈希表和一个堆。

哈希表中用来哈希的关键字是数据包的源/目的IP、源/目的端口和TCP/UDP（以下简称五元组）。哈希表节点中存储的信息是五元组、该条目第一次出现时的时间戳、该条目上一次出现的时间戳t、条目出现的次数c和对应的堆下标。

堆中存储的信息是指向哈希表中对应节点的指针。堆的排序关键字是估值函数f(c,t)。

### 5.2.2算法操作流程

算法中包含的操作有记录新数据包、删除旧条目、输出条目和计算估值函数。

#### 5.2.2.1记录新数据包

这个操作用来处理新抓取的数据包信息。传入参数是五元组和时间戳。具体操作为：

在哈希表中查询五元组，（1）如果存在，则更新该条目的出现次数和时间戳，检查时间戳与第一次出现时间戳的间隔是否超过上限，如果是，则删除这个条目，否则重新调整堆。（2）如果不存在，则在哈希表中添加一个新条目，并在堆中添加一个新元素，重新调整堆。检查是否到达哈希表容量上限，如果到达则执行“删除条目”操作来删除堆顶元素。

伪代码为：

NEW-PACKET(timestamp, quintuple)

if quintuple exists in hash

hash[quintuple].count = hash[quintuple].count + 1

hash[quintuple].timestamp = timestamp

if (timestamp – hash[quintuple].first\_timestamp) >= TIME\_LIMIT

DELETE-ELEMENT(hash[quintuple].heap)

else

adjust\_heap(hash[quintuple].heap)//parameter is heap index

else

hash[quintuple] = new hash

hash[quintuple].count = 1

hash[quintuple].timestamp = timestamp

hash[quintuple].first\_timestamp = timestamp

add element at the end of heap

hash[quintuple].heap = heap\_end//index of last heap element

heap[heap\_end].hash = &hash[quintuple]//hash node pointer

adjust\_heap(heap\_end)

if hash.size >= HASH\_SIZE\_LIMIT

DELETE-ELEMENT(0) //delete heap top element

#### 5.2.2.2删除条目

这个操作用来删除一个条目。具体操作为：

输出这个条目。删除这个条目在堆中的节点。调整堆。删除这个条目在哈希表中的节点。

伪代码：

DELETE-ELEMENT(heap\_index)

hashp = heap[heap\_index].hash//hash node pointer

PRINT(\*hashp)

heap[heap\_index] = heap[heap\_end--]

adjust\_heap(heap\_index)

delete hashp from hash

#### 5.2.2.3输出条目

用来输出一个条目。具体操作为：

输出五元组、第一次时间戳、最后一次时间戳。

伪代码：

PRINT(hash\_node)

print hash\_node. quintuple

print hash\_node.first\_timestamp

print hash\_node.timestamp

#### 5.2.2.4计算估值函数

这个操作计算估值函数，用来维护堆。调整堆的时候被调用。估值函数有两个参数，分别是条目出现的总次数和条目上一次出现到现在的时间间隔。估值函数形式如下：



其中time的单位是微秒。

伪代码为：

EVALUATE(hashp, now)

return hashp->count – (hashp->timestamp – now) / 1000

## 5.3复杂度分析

当哈希表的装填因子较小时，在哈希表中添加、查找、删除一个元素的平均时间复杂度为O(1)。在堆中添加、修改、删除一个元素的平均时间复杂度为O(log(n))，其中n为堆中元素个数。计算估值函数的时间复杂度为O(1)。

所以，这个合并算法处理一个数据包的时间复杂度为O(log(max))，其中max为算法中设置的哈希表元素个数上限。

哈希表的空间复杂度为O(n)，堆的空间复杂度为O(n)，其中n为哈希表中的元素个数。

所以，这个合并算法的空间复杂度为O(max)，其中max为算法中设置的哈希表元素个数上限。

当max设置为实际应用中的典型值时，处理一个数据包的时间远小于抓取这个数据包的时间，所需的空间也在允许范围内，所以时间复杂度和空间复杂度都比较理想。

# 系统详细设计与实现

## 6.1系统功能和架构

我们的目标系统是一套虚拟化环境下的分布式高效数据包信息采集与记录系统。这套系统要能够分布式地部署在虚拟化环境中，高效地采集网络流量中的关键信息，并且把信息集中记录下来。它要满足的一些要求包括：

1. 能够高效地采集主机、网络设备上的网络流量，提取时间戳、源/目的IP地址、源/目的端口和TCP/UDP，并进行记录；
2. 采集数据包时能够按照设定的规则进行数据包的过滤；
3. 能够在记录数据之前对重复数据进行合并，以减少存储空间的开销；
4. 能够分布式地部署，覆盖数据中心指定的区域；
5. 数据被记录下来之后能够集中起来存储；
6. 数据包采集程序能够统一控制和管理，数据包的采集规则能够统一下发。

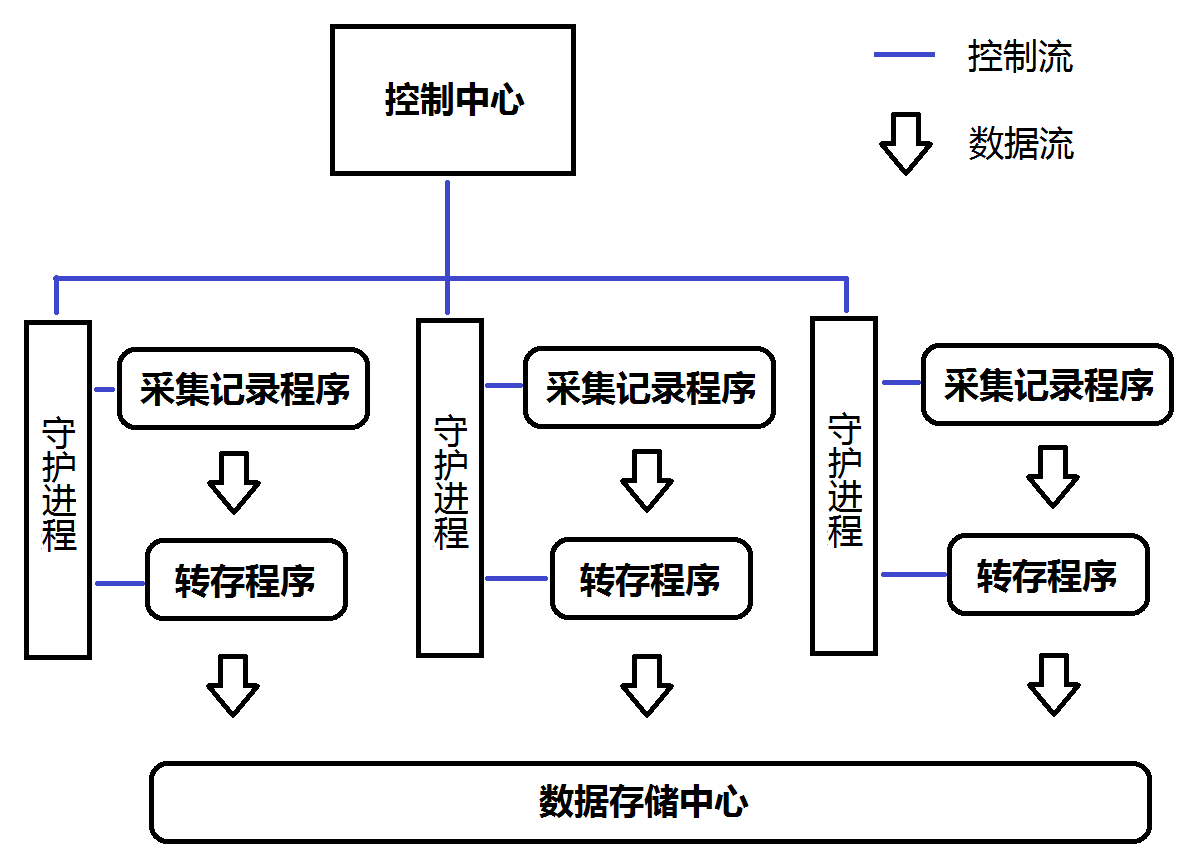


图6-1系统架构

为了满足上述功能需求，我们采用分布式架构。在每一台需要覆盖的主机上和网络设备端口镜像的专用处理服务器上安装数据包采集记录程序。主机上使用libpcap方案，端口镜像上使用多进程libpcap方案。采集到的数据先存储到本地文件，再经过另一个转存程序转发到集中存储的主机上。每个数据包采集记录程序和转存程序的配置、启动、停止都由一个守护进程来控制。另外，有一个控制中心，用户可以通过控制中心与每台主机上的守护进程通信，对数据包采集记录程序和转存程序进行配置变更、启动、停止等控制操作。

## 6.2数据包采集记录程序

数据包采集记录程序的功能是在特定端口根据包过滤规则抓取需要的数据包，并且进行协议解析，提取出需要的字段，将重复的条目合并，并且输出到文件中。

数据包采集记录程序分布式地部署在所有需要监控的主机和处理端口镜像的专用服务器上。

采集记录程序从配置文件和命令行参数中读取配置。配置文件中的参数包括要主机编号、监听端口、包过滤规则和需要解析的应用层协议。命令行参数中包含补充包过滤规则和补充监听端口。这两个补充参数是用来实现多进程libpcap方案而设置的，用来进行负载均衡和多端口同时抓取。

采集记录程序分为初始化、协议解析、数据包合并、输出四个模块。其中数据包合并模块已经在第五章进行了详细介绍。输出模块就是把合并后的条目按行输出到指定文件。所以不再介绍这两个模块。

### 6.2.1初始化模块

初始化模块负责读取配置、处理命令行参数、初始化内部变量、启动抓包循环等工作。

1. 首先从配置文件中读取主机编号、监听端口、包过滤规则和需要解析的应用层协议。
2. 从命令行参数中读取补充包过滤规则。如果该规则不为空，则将包过滤规则和补充包过滤规则用“与”操作符拼接起来作为新的包过滤规则，并且读取补充监听端口。如果补充监听端口也不为空，则用补充监听端口替代监听端口。
3. 初始化日志文件指针，初始化协议端口快速查找表。
4. 设置libpcap参数，编译并设置bpf包过滤规则，启动libpcap抓包循环。

在启动抓包循环时，将协议解析模块对应的函数指针传递给了libpcap，之后每抓取到一个包都会调用一次这个函数，对抓取到的数据包进行处理。

### 6.2.2协议解析模块

协议解析模块用来对数据包进行协议解析，自下而上解析Ethernet、VLAN、IP、TCP、UDP、HTTP、FTP等协议，提取出源/目的IP地址、源/目的端口和TCP/UDP，对HTTP数据包，提取出URI，对FTP数据包，提取FTP命令。

1. 从Ethernet报头中读取上层协议号，如果协议号为VLAN over Ethernet（0x8100），则继续执行步骤2，如果协议号为IP over Ethernet（0x0800）则跳到步骤3，否则丢弃这个数据包并结束解析。
2. 从VLAN协议段中读取上层协议号，如果协议号为IP over Ethernet（0x0800），则将IP协议初始偏移量增加4，并继续执行步骤3，否则丢弃这个数据包并结束解析。
3. 以IP协议初始偏移量为基准，根据IP协议规范（RFC 791）中规定的字段偏移量，从IP报头中读取源IP地址、目的IP地址、上层协议号、IP报头长度和IP分段标志。
4. 如果IP上层协议号既不是TCP over IP（6）也不是UDP over IP（17），或者设置了IP分段标志，则丢弃这个包并结束协议解析，否则继续执行步骤5。
5. 将IP协议初始偏移量与IP报头长度相加，得到TCP/UDP初始偏移量。如果IP上层协议号为UDP over IP（17）则继续执行步骤6，否则执行步骤7。
6. 根据TCP/UDP初始偏移量和UDP协议标准（RFC 768）中规定的字段偏移量，读取源端口、目的端口，并将TCP/UDP初始偏移量与UDP报头长度相加，得到应用层负载偏移量。结束协议解析。
7. 根据TCP/UDP初始偏移量和TCP协议标准（RFC 791）中规定的字段偏移量，读取源端口、目的端口和TCP报头长度，并将TCP/UDP初始偏移量与TCP报头长度相加，得到应用层负载偏移量。
8. 根据TCP端口号和协议端口快速查找表确定数据包是否可能是FTP请求数据包或HTTP请求数据包。如果可能是FTP请求数据包，则执行步骤9，如果可能是HTTP请求数据包，则执行步骤10，否则结束协议解析。
9. 根据FTP协议规范（RFC 959）中规定的FTP通信格式，提取FTP命令。结束协议解析。
10. 根据HTTP协议规范（RFC 2616）解析HTTP协议。从第一行提取方法，如果不是“GET”也不是“POST”，则结束协议解析。否则从第一行提取原始URI。如果URI以斜杠“/”开头，则此URI为相对地址，继续从后面的HTTP头中读取HOST字段，将HOST字段拼接在原始URI前面，得到URI。结束协议解析。

如果在协议解析过程中出现错误或者数据包属于不需要解析的协议类型，则丢弃数据包并结束对这个数据包的处理。

否则，将解析得到的字段传递给数据包合并模块。

## 6.3转存程序和数据存储中心

转存程序的功能是，每隔一定时间检查一次本地的数据文件，如果有采集记录程序更新的新数据，则读取这部分更新，并且将它们转发到到数据存储中心。

转存程序与采集记录程序配套部署在所有需要监控的主机和处理端口镜像的专用服务器上。所有转存程序可以把数据转发给同一个数据存储中心，也可以先转发到分中心，再集中到一个数据存储中心。

数据存储中心可以是数据库、文件接收器或者专用的客户端程序。数据存储中心的实现方式决定了数据转发的方式。

这两个部分的组合方式比较灵活，而且也没有需要特别说明的地方，所以具体实现就不再介绍。

## 6.4守护进程和控制中心

守护进程的功能是，作为一个linux守护进程稳定地运行并实现开机启动。在此基础上，以创建子进程的方式启动采集记录程序和转存程序，控制它们的启动、停止和配置。另外，向控制中心开放接口，是控制中心能够远程控制分布式部署的程序。

1. 通过系统调用把进程设置为守护进程。
2. 从配置文件中要监听的端口列表和TCP/UDP多进程的数量，并据此生成子进程的命令行参数列表。
3. 尝试启动所有子进程。如果有任何一个子进程启动失败，则在日志中记录错误信息。
4. 申请TCP端口，开始监听配置变更请求。
5. 等待下一个请求到来。接受请求，进行身份验证与数据校验。如果失败，则在日志中记录错误信息，然后重新执行本步骤。否则继续执行步骤6。
6. 把新配置写入配置文件。
7. 向所有子进程发送终止运行信号。等待一段之间后，释放子进程。
8. 尝试启动所有子进程。如果有任何一个子进程启动失败，则在日志中记录错误信息，并且向控制中心返回“配置更新失败”，否则向控制中心返回“配置更新成功”。跳到步骤5。

控制中心的功能就是将用户输入的新配置通过TCP通信发送到守护进程，并且把结果返回给用户。

# 系统实验

## 7.1实验环境

实验用的系统部署在一个基于OpenStack的实验性云计算平台上。

我们在两台物理主机上部署了单进程数据包采集记录程序、转存程序和守护进程。在一台交换机上设置了端口镜像，并且把镜像流量发送给一台配备有万兆网卡的专用服务器。在这台专用服务器上部署了多进程数据包采集记录程序、转存程序和守护进程。在两台主机上分别部署了数据存储中心和控制中心。

## 7.2功能验证

系统成功采集并记录了需要的数据包信息。样例如下：

#TCP`1412596446`0a64f94d`191d`0a0a5a80`0016`430`

#TCP`1412596448`0a0a5a80`0016`0a64f94d`191d`377`

#UDP`1412596449`0a0a5ab4`fbb9`0a0a5aff`00b1`11`

#UDP`1412596451`0a0a5ab4`dfdd`0a0a5aff`076c`28`

#UDP`1412596456`0a0a5a80`4162`ca78e006`0035`1`

#HTTP`TCP`1412596456`0a0a5a80`acaa`b73d092d`0050`http://www.bilibili.com/`

#HTTP`TCP`1412596464`0a0a5a80`8ff2`3dacc915`0050`http://www.sina.com.cn/`

#FTP`TCP`1412596737`0a0a5a80`af39`0a6c0afc`0015`USER ftp`

#FTP`TCP`1412596738`0a0a5a80`af39`0a6c0afc`0015`PASS ftp`

以TCP、UDP开头的是普通的TCP、UDP包，它们的格式是

‘#’ <’TCP’或’UDP’> ‘`’ <时间戳> ‘`’ <源IP地址> ‘`’ <源端口> ‘`’ <目的IP地址> ‘`’ <目的端口> ‘`’ <出现次数> ‘`’

其中IP地址和端口都用十六进制表示。

例如，其中第一条表示的意思是：

2014/10/6 19:54:06左右，从10.100.249.77的6429端口向10.10.90.128的22端口发送了430个TCP包。

以HTTP、FTP开头的是包含HTTP URI和FTP命令的数据包，它们的格式与TCP/UDP数据包类似。它们的最后一个字段分别是HTTP URI和FTP命令。

尝试从控制中心向三台机器上部署的守护进程发送指令。分别发送启动、停止、更改配置指令。返回结果均为成功。查看目标机器的进程表和配置文件，能够看到进程按照指令启停，配置文件更新。

尝试产生从各种位置发送到各种位置的数据包，测试系统对虚拟化环境的适应性。这个实验与3.4节的实验类似，这里不再赘述。结果显示系统能够抓取到所有数据包。

## 7.3性能测试

我们分别对部署在物理主机和专用服务器上的子系统进行了性能测试，主要测试单进程和多进程采集记录系统的性能。

测试的方法与4.1节中的方法相似，所以这里不再赘述。测试方法的唯一不同之处在于，这次进行的测试统计了测试开始前和结束后网卡接收和发送的数据包数量，与系统抓获的数据包数量进行比较。

两次测试使用的文件大小不同，CPU使用时间不具有可比性，所以实验结果中只列出平均CPU使用率。

单进程采集记录程序：

抓取所有IP数据包。

网络流量稳定在124.7MB/s（998Mbps），平均CPU使用率79.2%。

网卡发送和接收的数据包总数为2059144，测试系统采集到的数据包总数为2058607。

多进程采集记录程序：

配置为TCP四个进程，UDP一个进程。抓取所有IP数据包。

网络流量在265MB/s（2120Mbps）左右浮动，四个抓取TCP的进程的平均CPU使用率为51.6%、49.3%、49.4%、49.3%。抓取UDP的进程的平均CPU使用率为0.0%。

网卡发送和接收的数据包总数为8037088，测试系统采集到的数据包总数为8036262。

实验结果显示，单进程采集记录程序能够用于满负载运行的千兆网卡。多进程采集记录程序能够应对2Gbps的流量。

## 7.4数据包信息合并测试

数据包信息合并功能的测试需要在目标环境中进行才能获得最真实的数据，因为影响信息合并效果的主要因素是数据包数量在各条数据连接上的分布以及互相之间出现时间的先后关系。因为这些分布特性非常复杂，规律性不强，所以很难在实验室生成测试用的网络流量。

但是因为这个功能是银联项目结束之后开发的新功能，所以已经无法再拿到原来的目标环境进行测试。

最终，我们采取的方法是，使用旧版本系统在银联测试时生成的数据。因为旧版本系统没有合并功能，所以这些数据都是未经合并的数据包信息。使用这些数据作为输入，就可以得到比较真实的测试数据。

具体的实验方法是，把未经合并的数据存入文件。保持这些数据的顺序与数据包被抓取的顺序相同。抽取出系统中数据包信息合并相关的代码，改写成一个独立的信息合并程序。运行这个程序，并统计处理前后数据的条数和大小。

这里所说的未合并的数据指TCP/UDP条目。对HTTP、FTP条目不进行合并，所以输入文件中这些条目都被去除了。

实验结果：

处理前数据量为279859824条，数据文件大小为11.7GB。

处理后数据量为6336722条，数据文件大小为272MB。

实验结果显示，数据合并模块能够实现高效的信息合并，合并前后的数据条数为44.1:1。

# 总结与展望

本文对虚拟化环境下的数据包抓取和记录问题进行了研究，面对数据包抓取点选择、数据包抓取方案、数据包信息合并等问题，提出了自己的见解和解决方案，最终设计和实现了虚拟化环境下的分布式高效数据包信息采集与记录系统，并对系统进行了一系列测试。

在数据包抓取点的研究中，针对虚拟化网络的特有架构进行数据流向的研究，探索出了一套可以完整抓取所有数据包的抓取点组合方案。

在数据包抓取方案的研究中，尝试了几种可能的解决方案，分析了它们性能上的差别以及产生这些差别的原因，最终选择以libpcap为基础的方案，并提出了一种基于bpf流量负载均衡的多进程libpcap方案。

在数据包信息合并的研究中，针对问题的特点提出了一种由哈希表、堆排序和估值函数有机组合而成的合并算法，达到了比较理想的信息合并压缩效果。

并且最终根据上述研究和探索得到的结论，设计并实现了目标系统，对系统进行了一系列测试和验证，得到了比较满意的结果。

但是，由于时间和个人能力所限，本文的研究也存在很多不完善的地方，值得在未来的研究中进一步补充和加强。比如：

为了进一步提高数据包抓取效率，应当更深入地研究系统内核和网卡驱动的实现方式，通过编写专用网卡驱动和改写内核中相应的机制来进一步提高效率。

从更多维度考虑数据包信息合并和去重的工作，比如，有些数据包会在几个地方被抓去到，去除不同地方抓取到的数据中的重复内容也可以实现数据存储效率的很大提升。比如，从更大的时间尺度上总结数据包出现的规律，这样，符合规律的数据包就不需要在记录，只要记录违反规律的数据包就行了。

在虚拟网络中，数据包的IP地址往往会经过转换。如果能够根据IP地址转换表和抓取数据包的位置，把一个数据包对应的多个IP地址标注在采集的信息上，就能够更好地帮助用户或上层应用理解获取的数据。

# 参考文献

[1] Francesco Fusco, Luca Deri. High speed network traffic analysis with commodity multi-core systems [A]. In IMC '10 Proceedings of the 10th ACM SIGCOMM conference on Internet measurement [C]. New York, NY, USA: ACM, 2010: 218-224.

[2] Spring, J. Monitoring Cloud Computing by Layer, Part 1 [A]. In Security & Privacy, IEEE (Volume:9 , Issue: 2 ) [C]. IEEE, 2011: 66-68.

[3] Wen Shuguang, Xie Gaogang. libpcap-MT: A General Purpose Packet Capture Library with Multi-Thread [J]. In Journal of Computer Research and Development, 2011, 05.

[4] Giorgos Vasiliadis, Spiros Antonatos. Gnort: High Performance Network Intrusion Detection Using Graphics Processors [A].In Recent Advances in Intrusion Detection [C] 2008: 116-134.

[5] Snort Project. <http://www.openstack.org/>

[6] Libpcap Project. <http://www.tcpdump.org/>

[7] Linux kernel. <https://www.kernel.org/>

[8] IETF. Internet Protocol [EB/OL]. IETF RFC 791, 1981.

[9] IETF. Transmission Control Protocol [EB/OL]. IETF RFC 793, 1981.

[10] IETF. User Datagram Protocol [EB/OL]. IETF RFC 768, 1980.

[11] IETF. Hypertext Transfer Protocol -- HTTP/1.1 [EB/OL]. IETF RFC 2616, 1999.

[12] IETF. File Transfer Protocol [EB/OL]. IETF RFC 959, 1985.

[13] IETF. Defending Against Sequence Number Attacks [EB/OL]. IETF RFC 1948, 1996.

[14] Linux kernel. Packet Mmap [EB/OL].

<https://www.kernel.org/doc/Documentation/networking/packet_mmap.txt>

[15] Linux Kernel. Linux Socket Filtering aka Berkeley Packet Filter [EB/OL].

<https://www.kernel.org/doc/Documentation/networking/filter.txt>

[16] Linux Foundation. NAPI [EB/OL].

<http://www.linuxfoundation.org/collaborate/workgroups/networking/napi>

[17] Vimalkumar Jeyakumar, Mohammad Alizadeh. EyeQ: Practical Network Performance Isolation at the Edge [A]. In 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI ’13) [C]. 2013:297-311.

[18] Vermeiren, T., Borghs, E., Haaodorens, B. Evaluation of software techniques for parallel packet processing on multi-core processors [A]. In Consumer Communications and Networking Conference [C]. IEEE, Las Vegas, NV, USA, 2004: 645-647.

[20] Zhifeng Xiao. Security and Privacy in Cloud Computing [A]. In Communications Surveys & Tutorials [C]. IEEE, 2013: 843-859.

[21] Nair, S.K, Porwal, S. Towards Secure Cloud Bursting, Brokerage and Aggregation [A]. In 8th European Conference on Web Services [C]. IEEE, 2010: 189-196.

[22] Zhen Ling, Xinwen Fu. Novel Packet Size-Based Covert Channel Attacks against Anonymizer [A]. In IEEE Transactions on Computers [C]. IEEE, 2013: 2411-2426.

[23] Lawey, A.Q., El-Gorashi, T.E.H. Distributed Energy Efficient Clouds Over Core Networks [A]. In Journal of Lightwave Technology [C]. IEEE, 2014: 1261-1281.

[24] Smallwood D., Vance A. Intrusion analysis with deep packet inspection: Increasing efficiency of packet based investigations [A]. In International Conference on Cloud and Service Computing [C]. IEEE, 2011: 342-347.

[25] Varadharajan, V., Tupakula, U. Security as a Service Model for Cloud Environment [A]. In IEEE Transactions on Network and Service Management [C]. IEEE, 2014: 60-75.

[26] Agrawal, D., Das, S. Data Management in the Cloud:Challenges and Opportunities [A]. IEEE, 2012.

[27] Thorpe, S. Towards a Forensic-Based Service Oriented Architecture Framework for Auditing of Cloud Logs [A]. In Ninth World Congress on Services [C]. IEEE, 2013:75-83.

[28] Ying-dar Lin. On-the-Fly Capture and Replay Mechanisms for Multi-Port Network Devices in Operational Networks [A]. In IEEE Transactions on Network and Service Management [C]. IEEE, 2014:158-171.

[29] Shetty, S. Auditing and Analysis of Network Traffic in Cloud Environment [A]. In IEEE Ninth World Congress on Services [C]. IEEE, 2013: 260-267.

[30] Qiang Guan. CDA: A Cloud Dependability Analysis Framework for Characterizing System Dependability in Cloud Computing Infrastructures [A]. In IEEE 18th Pacific Rim International Symposium on Dependable Computing [C]. IEEE, 2012: 11-20.

[31] Yegenoglu, F. A model for representing wide area Internet packet behavior [A]. In Performance, Computing, and Communications Conference [C]. IEEE, 2000: 167-173.

[32] Lin, Y.-D. PCAPLib: A System of Extracting, Classifying, and Anonymizing Real Packet Traces [A]. In Systems Journal [C]. IEEE, 2014: 1-12.

[33] Nelms, T. Packet scheduling for deep packet inspection on multi-core architectures [A]. In ACM/IEEE Symposium on Architectures for Networking and Communications Systems [C]. IEEE, 2010: 1-11.

[34] Zhanikeev, Marat. Experiments with practical on-demand multi-core packet capture [A]. In 15th Asia-Pacific Network Operations and Management Symposium [C]. IEEE, 2013: 1-3.

# 后记

写毕业论文的过程就是学习、探索新知识，发现自己的不足的过程。在论文的准备和写作过程中，通过阅读大量的文献和资料，进行许多实验，大大加深了我对很多技术问题的认识，也让我意识到自己在很多方面知识的欠缺和研究态度、研究方法上的不足。

同时，这也是对我研究生期间工作的一个总结。在仔细梳理过去的工作的过程中，往往会发现需要纠正和改进的地方。这个过程是发人深省的，也是引人回忆的。

在这里，我要感谢实验室导师组在这几年中对我的指导和启发。感谢我的导师吴承荣副教授对我的培养。感谢张世永教授、钟亦平教授平时的教导和关怀。感谢吴杰教授、吕智慧副教授、叶家炜老师的指导和支持。感谢实验室对我各方面的支持和理解。