

华中科技大学网络空间安全学院

《网络安全课程设计》实验报告  
——Linux 下状态检测防火墙的设计与实现

姓 名 郭倜维

学 号 U201614848

班 级 IS1602

指导教师 肖凌

2020 年 1 月 7 日

## 目 录

一、实验目的和要求.....	2
二、实验原理（简述）.....	3
三、实验环境和采用的工具.....	6
四、系统设计.....	6
五、系统详细设计.....	9
六、系统测试.....	14
七、心得体会.....	17

## 一、实验目的和要求

计算机网络安全是信息安全专业的一门核心课程，网络安全课程设计是计算机网络安全课程的一个综合实践环节。通过本课程设计要使学生达到以下目标：

- (1) 结合理论课程学习，深入理解计算机网络安全的基本原理与协议，巩固计算机网络安全基本理论知识；
- (2) 熟练掌握计算机网络编程方法，拓展学生的应用能力；
- (3) 加强对网络协议栈的理解；
- (4) 提高分析、设计软件系统以及编写文档的能力；
- (5) 培养团队合作能力。
- (6) 通过学习和了解 Linux Netfilter 架构和原理，了解 Linux 的网络协议栈的处理过程；
- (7) 利用 Linux Netfilter 架构，编写一个具有状态检测的 linux 防火墙，能对收发的报文进行状态分析和过滤；
- (8) 实验验证阶段需要综合运用网络组网技术、网络协议分析技术；
- (9) 通过 Linux 下的开发过程提升学生的内核编程能力和分析问题、解决问题的能力；
- (10) 撰写课程设计报告；
- (11) 实验结果展示。

课程设计的防火墙系统包括两个部分，一个是防火墙内核模块，一个是应用程序。内核模块用来截获通过防火墙的报文，进行规则过滤，根据规则来决定是禁止还是放行，若放行，是直接路由转发还是 NAT 之后转发，维护连接表。规则由应用程序通过跟内核的接口写入；

应用程序用来设置规则（向内核模块写入），查看日志（从内核模块获得，内核模块不要直接写文件，效率低），查看连接（从内核模块获得）等操作，需要跟内核模块之间通过一定的接口来进行数据交换。

### (1) 系统运行

系统启动以后插入模块，防火墙以内核模块方式运行；

应用程序读取配置，向内核写入规则，报文到达，按照规则进行处理；

### (2) 界面

采用图形或者命令行方式进行规则配置，界面友好；

### (3) 功能要求

能对 TCP、UDP、ICMP 协议的报文进行过滤

每一条 过滤规则至少包含：报文的源 IP(带掩码的网络地址)、目的 IP(带掩码的网络地址)、源端口、目的端口、协议、动作（禁止/允许），是否记录日志；

过滤规则可以进行添加、删除、保存，配置的规则能立即生效；  
过滤日志可以查看；  
具有 NAT 功能，转换地址分按接口地址转换和指定地址转换（能实现源或者目的地址转换的任一种即可）；  
能查看所有连接状态信息。

#### (4) 测试

测试系统是否符合设计要求

系统运行稳定

性能分析

## 二、实验原理（简述）

### 1. 状态检测防火墙的工作原理

状态检测表包括所有通过防火墙的连接，并维护所有连接每条连接的信息包括源地址、目的地址、协议类型、协议相关信息（如 TCP/UDP 协议的端口、ICMP 协议的 ID 号）、连接状态（如 TCP 连接状态）和超时时间（若进行 NAT 转换，还需记录转换前后地址端口信息）等，防火墙把这些信息叫做状态。通过状态检测，可实现比简单包过滤防火墙更大的安全性及更高的性能。

在建立非 TCP 的连接时，防火墙为其建立虚拟连接，如 UDP 时，一方发出 UDP 包吼，如 DNS 请求，防火墙会将从目的地址和端口返回的，到达源地址源端口的包作为状态相关的包而允许通过。在建立 ICMP 时，获得信息查询报文吼，如 ping 包，其状态相关包就是来自目的地址的 echo reply 包，而没有与 echo 包对应的 echo reply 包则认为时状态非法的。这些链接在防火墙的状态检测表中的超时时间比较短。

状态检测防火墙基本保持了简单包过滤防火墙的优点，性能比较好，同时对应用是透明的，在此基础上，对于安全性有了大幅提升。这种防火墙摒弃了简单包过滤防火墙仅仅考察进出网络的数据包，不关心数据包状态的缺点，在防火墙的核心部分建立状态连接表，维护了连接，将进出网络的数据当成一个个的事件来处理。可以说，状态检测包过滤防火墙规范了网络层和传输层行为，而应用代理型防火墙则是规范了特定的应用协议上的行为。

### 2. Linux Netfilter 架构

Netfilter/IPTables 是 Linux2.4.x 之后新一代的 Linux 防火墙机制，是 linux 内核的一个子系统。Netfilter 采用模块化设计，具有良好的可扩充性。其重要工具模块 IPTables 从用户态的 iptables 连接到内核态的 Netfilter 的架构中，Netfilter 与 IP 协议栈是无缝契合的，并允许使用者对数据报进行过滤、地址转换、处理

等操作，其具体功能模块有连接跟踪模块，网络地址转换模块，数据包修改模块，其他高级功能模块。

Netfilter 主要通过表、链实现规则，可以说，Netfilter 是表的容器，表是链的容器，链是规则的容器，最终形成对数据报处理规则的实现。

数据在协议栈里的发送过程中，从上至下依次是“加头”的过程，每到达一层数据就被会加上该层的头部；与此同时，接受数据方就是个“剥头”的过程，从网卡收上包来之后，在往协议栈的上层传递过程中依次剥去每层的头部，最终到达用户那儿的就是裸数据了。

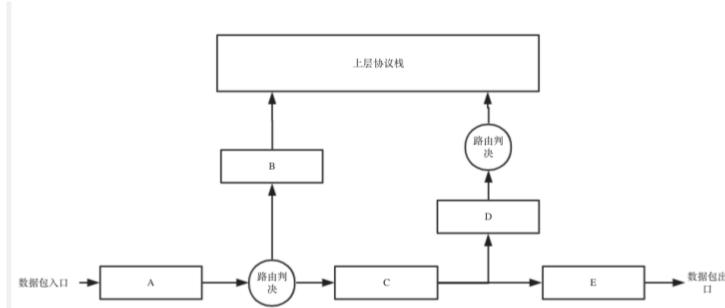


图 1 Netfilter 结构

对于收到的每个数据包，都从“A”点进来，经过路由判决，如果是发送给本机的就经过“B”点，然后往协议栈的上层继续传递；否则，如果该数据包的目的地是不本机，那么就经过“C”点，然后顺着“E”点将该包转发出去。对于发送的每个数据包，首先也有一个路由判决，以确定该包是从哪个接口出去，然后经过“D”点，最后也是顺着“E”点将该包发送出去。协议栈那五个关键点 A, B, C, D 和 E 就是我们 Netfilter 大展拳脚的地方了。

整个 Linux 内核中 Net filter 框架的 HOOK 机制可以概括如下图 2-2 所示，在数据包流经内核协议栈的整个过程中，在一些已预定义的关键点上 PRE\_ROUTING、LOCAL\_IN、FORWARD、LOCAL\_OUT 和 POST\_ROUTING 会根据数据包的协议簇 PF\_INET 到这些关键点去查找是否注册有钩子函数。如果没有，则直接返回 okfn 函数指针所指向的函数继续走协议栈；如果有，则调用 nf\_hook\_slow 函数，从而进入到 Netfilter 框架中去进一步调用已注册在该过滤点下的钩子函数，再根据其返回值来确定是否继续执行由函数指针 okfn 所指向的函数。

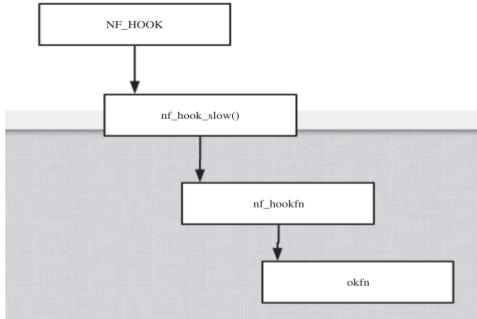


图 2 Netfilter 的 HOOK 机制

### 3. Linux 内核编程

Linux 操作系统和驱动程序运行在内核空间（比如：报文过滤的内核模块），应用程序（比如：配置规则的程序）运行在用户空间，两者不能简单地使用指针传递数据，因为 Linux 使用的虚拟内存机制，用户空间的数据可能被换出，当内核空间使用用户空间指针时，对应的数据可能不在内存中。

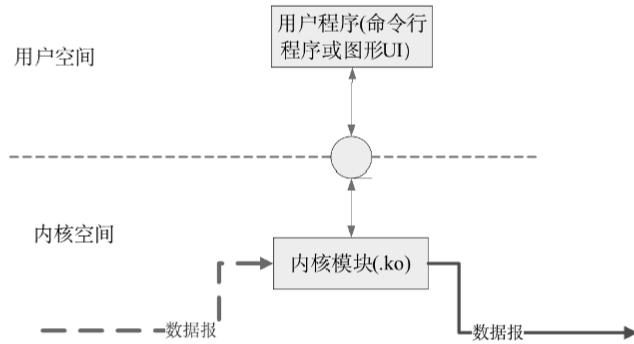


图 3 Linux 用户空间和内核空间模型

内核空间和用户空间信息交互的原理主要是通过以下方式进行的：

- 1) 编写自己的系统调用
- 2) 编写设备驱动程序

驱动程序运行于内核空间，用户空间的应用程序通过文件系统中/dev/目录下的一个文件（字符设备文件）来和它交互。这就是我们熟悉的那个文件操作流程：`open()` —— `read()` —— `write()` —— `ioctl()` —— `close()`。

驱动程序也是用户空间和内核信息交互的重要方式之一。其实 `ioctl`, `read`, `write` 本质上讲也是通过系统调用去完成的，只是这些调用已被内核进行了标准封装，统一定义。

- 3) 使用 proc 文件系统

`proc` 是 Linux 提供的一种特殊的文件系统，推出它的目的就是提供一种便捷的用户和内核间的交互方式。

- 4) 使用 netlink

### 5) 使用内存映像

内存影射方式通常也正是应用在那些内核和用户空间需要快速大量交互数据的情况下，特别是那些对实时性要求较强的应用。

## 三、实验环境和采用的工具

实验环境：Linux version 3.13.0-32-generic、Ubuntu12.04

虚拟机软件：Parallel Desktop 14

开发工具：qt-opensource-linux-x86-5.3.1.run

## 四、系统设计

### 1. 系统构成

系统主要包括两个部分，分别是内核部分和用户界面部分。其中，内核部分是通过字符设备实现的，而用户界面是通过 qt 实现的。

其中内核部分负责主要的数据包的过滤和处理，而用户 ui 则主要负责对内核部分的内容设置，以及将内核部分获得数据展现到用户界面中的效果。

#### 1) 内核部分

作为系统的核心模块的内核，需要实现包过滤功能、状态检测表功能、Nat 转换表功能以及日志的写入功能。

实现一个字符设备后，通过在字符设备中定义 hook 函数，实现 netfilter 的数据包过滤和处理。同时为了方便用户态能够获取字符设备中的内容和操作设备，需要给用户态预留足够的接口。

#### 2) 用户界面

用户界面可以完成对字符设备的一定操作和设置，字符设备链表中的数据在界面中用表格的形式展现出来，同时能开启模块中的 Nat 转换功能和日志写入功能；并能对状态链接表中的内容能够实现自动更新。

### 2. 系统框图

系统处理流程分为两步，一是处理单独的数据包，二是在开启了 Nat 转换后转化数据包。系统的整体框架如图 4 所示。

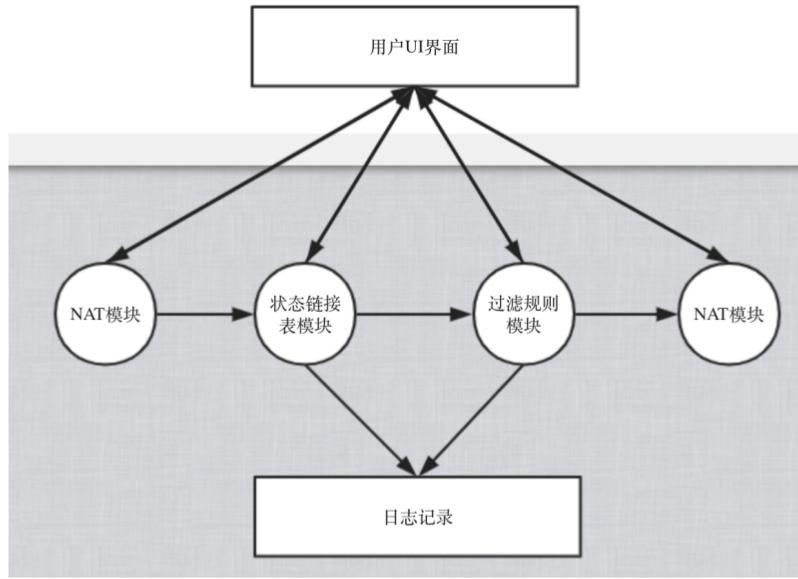


图 4 系统框架

### 3. 模块划分

本系统在 UI 用户端通过调用不同调用序号的 ioctl 函数来实现和内核中的不同函数的交互，从而达到访问不同模块的效果。

系统的主要模块分别为过滤规则表模块、状态检测表模块、Nat 转换表模块、日志模块和控制调用模块，具体实现原理如下。

#### 1) 过滤规则表模块

包过滤规则存放在一个单链表中，存放的是规则数据。通过直接顺序遍历的方法对包过滤进行检索，然后对过滤规则进行逐一匹配。链表实现框架如下所示。

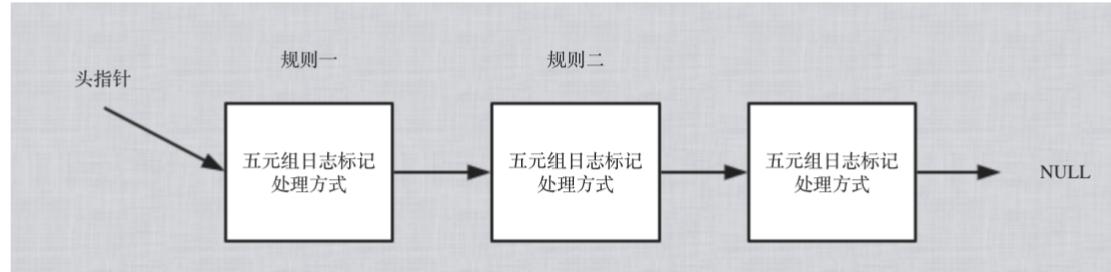


图 5 过滤规则表链表框架

#### 2) 状态检测表模块

状态检测表使用 hash 表来储存，从而可以通过 hash 搜索的方式达到快速检索的目的。定义一个大小为 1024 的数组，用其存放链表表头；对计算数据包中的 ip 地址和端口使用 hash 函数，得到一个散列值，将其存放到相应的表头的链表中。当 hash 链产生冲突时，通过链表的方式进行解决。hash 表框架如下所示。

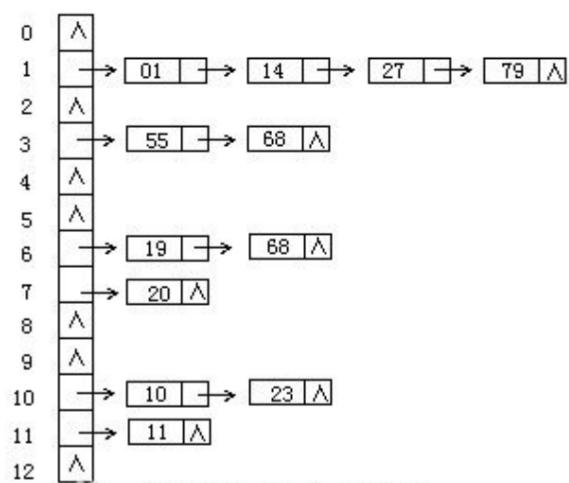


图 6 hash 表框架

### 3) Nat 转换表模块

实现动态分配端口的 Nat 转换表将已经存在的 Nat 转换通过链表储存起来。当开启 Nat 时，如果链接中的端口号和目的地址在 Nat 链表中不存在，则分配一个端口给内网中的主机去和外网进行数据转换。Nat 转换表框架如下所示。

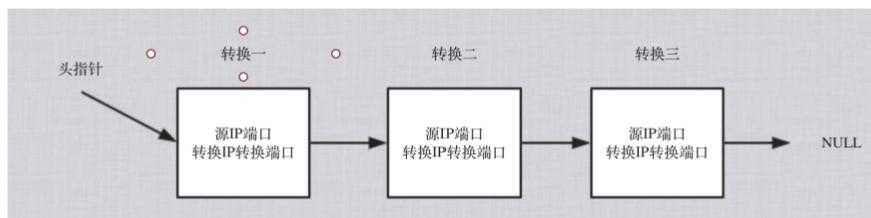


图 7 Nat 转换表框架

### 4) 日志模块

日志模块将日志内容存储在链表中。在用户态刷新获取内核中的内核日志数据后，将日志链表删除。

### 5) 控制调用模块

通过该模块，内核可以用 ioctl 函数来调用其他各个模块。调用该模块可以将用户态的数据输入到字符设备中，也可以将字符设备中的内容反馈给用户态。

## 4. 模块之间的接口

模块之间的功能调用以及用户态和内核的接口，主要包括以下几个功能：

- 1) 添加防火墙过滤规则
- 2) 删除防火墙过滤规则
- 3) 清空防火墙过滤规则
- 4) 获取状态链接表储存内容个数
- 5) 清空状态链接表
- 6) 获取 Nat 转换表长度
- 7) 清空 Nat 转换表规则清空
- 8) 开启 Nat 转换

- 9) 关闭 Nat 转换
- 10) 日志链表长度获取
- 11) 日志链表内容获取

## 五、系统详细设计

### 1. 关键模块的流程设计

#### 1) 处理数据包

数据包的处理调用了状态检测部分和包过滤部分，具体流程图如图 8 所示，即在 NF\_INET\_LOCAL\_IN、NF\_INET\_FORWARD 以及 NF\_INET\_LOCAL\_OUT 都挂载数据包过滤的钩子函数，以达到数据在结点流动过程的全覆盖处理的功能。

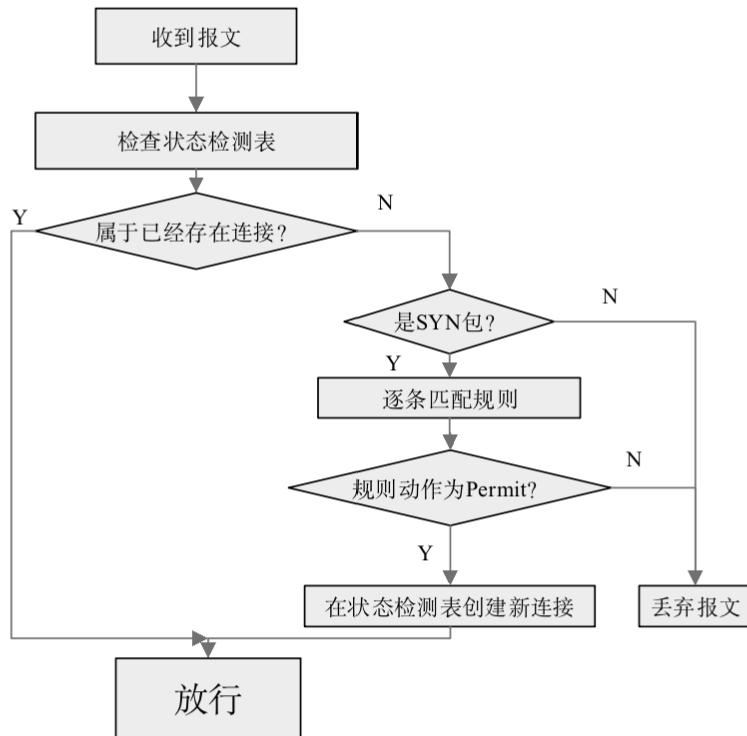


图 8 数据包处理流程

状态链接表部分，还需要设置一个定时器装置，用来对状态状态链接表中链接的生命时间进行更新。当状态链接表中的生命时间为 0 时删除该链接状态。单倘若该状态链接又出现了新的数据包的交换，则设置该状态链接的生命时间为最大值，表明该链接依然存在。

状态链接的判定，有两种，一是出现了符合包过滤规则的 SYN 包，则创建了一个新的状态链接；二是出现了符合包过滤规则的新的五元组内容，则创建一个新的状态链接。

## 2) NAT 转换

Nat 转换，是需要在数据报文进行处理之后，再对源地址进行修改的步骤，Nat 转换具体的流程如图 9 所示。因此，需要在 NF\_INET\_PRE\_ROUTING 和 NF\_INET\_POST\_ROUTING 结点挂载 Nat 转换的 hook 函数，这样才能顾及到数据包进入和发出的全过程中的 Nat 转换。

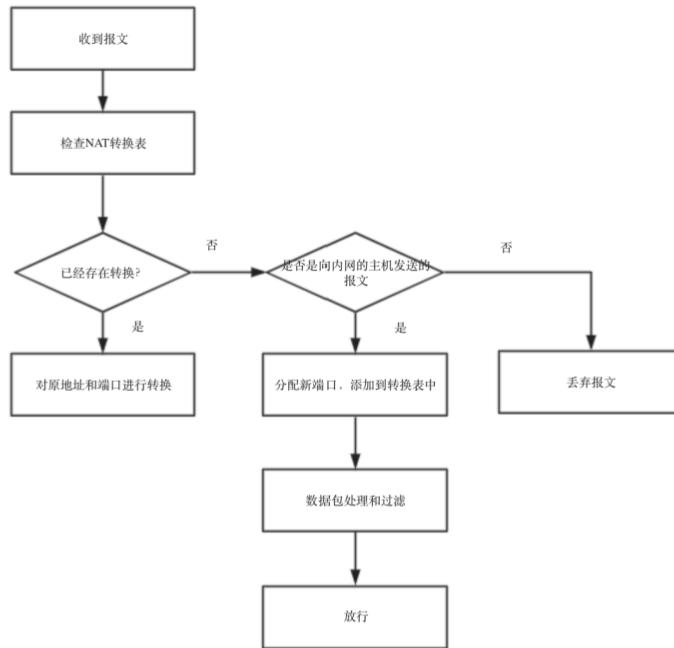


图 9 Nat 转换思路

## 3) 日志审计

记录动作的源 ip、目的 ip；源端口、目的端口；协议类型；年、月、日、时、分、秒；规则设置的动作等内容。

日志记录，是在包过滤和状态检测，没完成一次规则的匹配和处理时，就对该操作的一些信息进行一些记录，储存在日志的链表中。等待 UI 界面获取日志的内容，当 ui 获取了日志的内容后，日志链表中的内容将被删除。

## 4) UI 交互

该部分实现的核心，就是实现内核中的内容的获取和写入。通过调用内核模块预留的接口，实现用户态对内核部分的控制。

其基本的实现关键即 copy\_from\_user 函数和 copy\_to\_user 函数的使用。copy\_from\_user，可以将用户态的数据传输到内核态之中，因此 ui 界面中输入的过滤规则等内容，将会利用该函数传到内核态的规则链表中。copy\_to\_user 函数，可以将内核态的内容传递到用户态中，对于状态链接表链表中的内容以及 Nat 转换表中的内容还有日志记录表中的内容，传送到用户态的区域中，再经过用户态进行处理和操作。

用户态通过 qt 设置一个定时器，每秒刷新一次状态链接表中的内容和 Nat 转换表中的内容。基本的思路为，首先通过 ioctl 函数获取状态链接表和 Nat 转换表中当前储存的数据的个数，然后在用户态中定义相应大小的储存区域，将内

核态的数据安全的传送到用户态中， 用户态再通过表格的形式将数据表示出来。

## 2. 关键数据结构设计

### 1) 包过滤规则链表

包过滤规则链表的节点结构如下所示，包含了源 ip、目的 ip; 源端口、目的端口； 协议类型； 掩码地址； 操作动作等内容

```
typedef struct Rule{  
    unsigned int sip;  
    unsigned int dip;  
    unsigned short sport;  
    unsigned short dport;  
    unsigned short protocol;  
    unsigned short sMask;  
    unsigned short dMask;  
    bool accept;  
    bool log;  
    struct Rule *next;           //单链表的指针域  
}Rule;
```

### 2) 状态检测表

状态检测表的数据结构下所示，包含了源 ip、目的 ip; 源端口、目的端口； 协议类型； 创建时间； 生命时间； 是否记录日志标记等内容

```
typedef struct ActiveLink {  
    __be32 sip;  
    __be32 dip;  
    __be16 sport;  
    __be16 dport;  
    __u8 protocol;  
    mtime createtime;  
    __u8 lifetime;  
    bool log;  
    struct ActiveLink *next;  
}ActiveLink;
```

### 3) 日志链表

日志链表的数据结构如下所示，包含了源 ip、目的 ip; 源端口、目的端口； 协议类型； 事件事件； 操作动作

```
typedef struct Log{  
    __be32 sip;  
    __be32 dip;  
    __be16 sport;  
    __be16 dport;  
    __u8 protocol;  
    mtime time;  
    bool accept;
```

```

    struct Log *next;
}Log;

```

### 3. 模块详细接口

#### 1) UI 交互模块

对 ioctl 函数分配不同的操作标志号，如下所示。在用户态调用 ioctl 函数，会调用内核中的 netfilter\_cdev\_ioctl 函数，因此在用户态和内核态设置相对应的操作 cmd 号，这样就能实现内核和用户态的联动。

#define FW_ADD_RULE	0
#define FW_REMOVE_RULE	1
#define FW_CLEAR_RULE	12
#define FW_GET_NAT_LEN	3
#define FW_REFRESH_NAT_RULE	4
#define FW_START_NAT_TRANSFORM	5
#define FW_STOP_NAT_TRANSFORM	6
#define FW_GET_ACTIVELINK_LEN	7
#define FW_REFRESH_ACTIVELINK	8
#define FW_GET_LOG_LEN	9
#define FW_WRITE_LOG	10

在 netfilter\_cdev\_ioctl 函数中，通过 switch 对不同的操作符进行对应的操作，具体操作如下图 10 所示。

```

switch(cmd) {
    case FW_ADD_RULE:
        //printf("\nfw_ADD_RULE\n");
        copy_from_user(&rule,(struct Rule *)arg, sizeof(struct Rule));
        addRule(&rule);
        break;
    case FW_REMOVE_RULE:
        //printf("\nfw_DEL_RULE\n");
        copy_from_user(&rule,(struct Rule *)arg, sizeof(struct Rule));
        removeRule(&rule);
        break;
    case FW_CLEAR_RULE:
        //printf("\nfw_CLEAR_RULE\n");
        clearRuleList();
        break;
    case FW_GET_NAT_LEN:
        //printf("\nfw_GET_NAT_LEN\n");
        NATLen = NATList->ip;
        copy_to_user((int *)arg,&NATLen,sizeof(NATLen));
        break;
    case FW_REFRESH_NAT_RULE:
        //printf("\nfw_REFRESH_NAT_RULE\n");
        NATNode = NATList->next;
        while(NATNode){
            copy_to_user((struct NATRule*)arg + move++,NATNode,sizeof(struct
                NATRule));
            NATNode = NATNode->next;
        }
        break;
    case FW_START_NAT_TRANSFORM:
        //printf("\nfw_START_NAT_TRANSFORM\n");
        NATFlag = true;
        copy_from_user(&hostInfo,(struct HostInfo *)arg, sizeof(struct HostInfo));
        break;
    case FW_STOP_NAT_TRANSFORM:
        //printf("\nfw_STOP_NAT_TRANSFORM\n");
        NATFlag = false;
        clearNATRule();
        NATPort = NAT_PORT_START;
        break;
}

```

图 10 netfilter\_cdev\_ioctl 函数

其中，状态检测表、nat 转换表和日志链表的内容传送到用户态空间时，需要先计算链表中的数据个数，方便约定一个转换区域的大小。具体操作如下图 11 所示。

```

case FW_GET_ACTIVELINK_LEN:
    //printk("\nFW_GET_ACTIVELINK_LEN!\n");
    for(int i = 0;i < ACTIVE_LINK_HASH_TABLE_SIZE;i++){
        linkLen += ActiveLinklist[i].sip;
    }
    copy_to_user((int *)arg,&linkLen,sizeof(linkLen));
    break;
case FW_REFRESH_ACTIVELINK:
    //printk("\nFW_REFRESH_ACTIVELINK!\n");
    for(int i = 0;i < ACTIVE_LINK_HASH_TABLE_SIZE;i++){
        linkNode = ActiveLinkList[i].next;
        while(linkNode){
            copy_to_user((struct ActiveLink *)arg + move++,linkNode,sizeof(struct ActiveLink));
            linkNode = linkNode->next;
        }
    }
    break;
case FW_GET_LOG_LEN:
    //printk("\nFW_GET_LOG_LEN!\n");
    logLen = LogList->sip;
    copy_to_user((int *)arg,&logLen,sizeof(logLen));
    break;
case FW_WRITE_LOG:
    //printk("\nFW_WRITE_LOG!\n");
    logNode = LogList->next;
    while(logNode){
        copy_to_user((struct Log*)arg + move++,logNode,sizeof(struct Log));
        logNode = logNode->next;
    }
    clearLogList();
    break;
}

```

图 11

## 2) NAT 转换模块

在进行 Nat 转换时，不仅需要将原地址和端口进行更换，还要将校验和进行重新计算，进行重新计算的过程可以通过 csum\_tcpudp\_magic 和 ip\_fast\_csum 函数，可以将 tcp 报文和 ip 报文的校验和进行重置。而对于 udp 报文，则需要只需将校验和设为 0，则不会受到影响。具体操作如下图 12 所示。

```

if(iph->protocol == IPPROTO_TCP){
    tcph->check = 0;
    skb->csum = csum_partial((unsigned char *)tcph, tot_len -
        iph_len,0);
    | tcph->check = csum_tcpudp_magic(iph->saddr,
        iph->daddr,
        ntohs(iph->tot_len) -
        iph_len,iph->protocol,
        skb->csum);
    iph->check = 0;
    iph->check = ip_fast_csum(iph,iph->ihl);
}
else{
    iph->check = 0;
}

```

图 12

Nat 转换时，Nat 的开启和关闭，通过一个变量控制，当变量值为 1 时，运行 Nat 转换模块，当变量值为 0 时，跳过 Nat 转换模块。具体操作如下图 13 所示

```

/*-----*
 * Hook func to do src NAT transform
 */
unsigned int srcNATHookFunc(unsigned int hooknum,
    struct sk_buff *skb,
    const struct net_device *in,
    const struct net_device *out,
    int (*okfn)(struct sk_buff *));
unsigned int ret = NF_ACCEPT; // default policy
if(NATFlag)
{
    _be32 *sip,*dip;
    _be16 *sport,*dport;
    struct iphdr *iph = ip_hdr(skb);
    struct tcphdr *tcph;
    struct udphdr *udph;
    NATRule *NATNode,NATrule;
    sip = &(iph->saddr);
    dip = &(iph->daddr);
    switch(iph->protocol){
        case IPPROTO_TCP:
            tcph = (struct tcphdr)((_u8 *)iph + (iph->ihl << 2));
            sport = &(tcph->source);
            dport = &(tcph->dest);
            break;
        case IPPROTO_UDP:
            udph = (struct udphdr)((_u8 *)iph + (iph->ihl << 2));
            sport = &(udph->source);
            dport = &(udph->dest);
            break;
        default:
            return NF_ACCEPT;
    }
}

```

图 13

### 3) 状态检测模块

为了实现快速查找功能，使用 hash 搜索的方式实现，其中 hash 函数为数据报文中的原地址和目标地址以及端口号乘以一个质数再模上 hash 表长度 1024 所得。hash 函数具体如下所示。其中为了让一个连接中的双方的发送报文都能定位到一个连接储存中，则将目的地址和原地址乘以的质数相同，同理端口号乘以的质数也相同。

```
unsigned int hashLink(const ActiveLink *link){  
    unsigned long long hash = 7;  
    hash += 13 * link->sip;  
    hash += 13 * link->dip;  
    hash += 19 * link->sport;  
    hash += 19 * link->dport;  
    hash += 29 * link->protocol;  
    return hash % ACTIVE_LINK_HASH_TABLE_SIZE;  
}
```

状态检测表的链接创建，有两个规则，一是 tcp 报文的 syn 请求报文，从内网中发送的，则创建一个状态链接，受到 fin 报文则删除该链接。二是受到符合包过滤条件的新的链接，则创建一个新的链接进状态链接表中。具体操作如下图 14 所示。

```
//sim.cs:doHandle  
if(rsLink){  
    printk("link established!\n");  
    if(protocol == IPPROTO_TCP){  
        if(tcph->fin){  
            removeActiveLink(&link);  
        }  
        logflag = rsLink->log;  
    }  
    else{  
        Rule *rule;  
        rule = matchRule(sip,sport,dip,dport,protocol);  
        if(rule){  
            printk("rule exists!\n");  
            if(rule->accept){  
                if(protocol == IPPROTO_TCP){  
                    if(tcph->syn){  
                        addActiveLink(&link);  
                    }  
                    else{  
                        ret = NF_DROP;  
                    }  
                }  
                else{  
                    addActiveLink(&link);  
                }  
            }  
            else{  
                ret = NF_DROP;  
            }  
            logflag = rule->log;  
        }  
        else{  
            ret = NF_DROP;  
        }  
    }  
}
```

图 14

## 六、系统测试

设置默认规则为拒绝，之后使用 ping 命令，此时 ping 不通。如图 15 所示。

```
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
^C
--- 8.8.8.8 ping statistics ---
4 packets transmitted, 0 received, 100% packet loss, time
```

图 15

首先允许所有的 ICMP 报文通过, 按照图 16 设置添加规则, IP 地址的 0.0.0.0 代表所有地址, 端口的 0 代表所有端口。添加成功后在右侧显示出当前所有规则。因为 ping 命令是建立在 ICMP 协议上的, 因此这时达到的效果是可以使用 ping 命令, 如图 17, ping 成功。

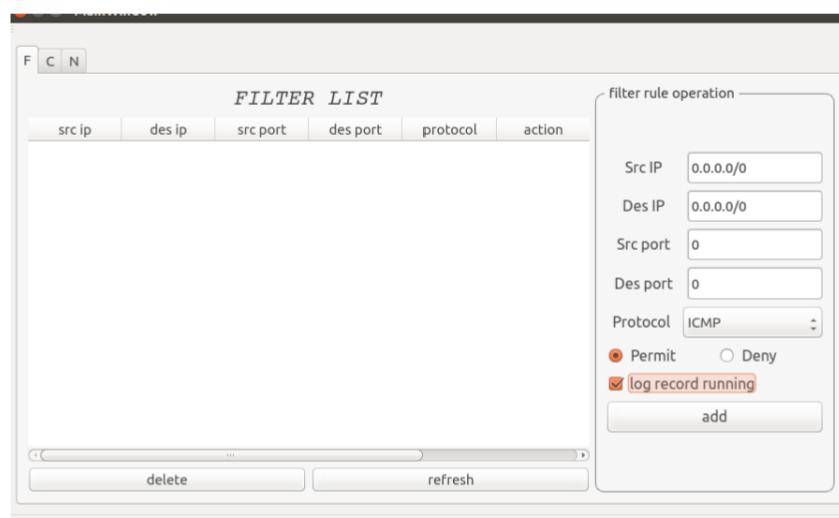


图 16 添加规则

```
root@ubuntu:/home/fhy# ping 8.8.8.8
PING 8.8.8.8 (8.8.8.8) 56(84) bytes of data.
ping: sendmsg: Operation not permitted
ping: sendmsg: Operation not permitted
^C
--- 8.8.8.8 ping statistics ---
2 packets transmitted, 0 received, 100% packet loss, time 1009ms
root@ubuntu:/home/fhy#
```

图 17 ping 成功

之后尝试 ping 命令对百度的域名解析 (如图 18), 结果失败。

```
root@ubuntu:/home/fhy# ping www.baidu.com
ping: unknown host www.baidu.com
root@ubuntu:/home/fhy#
```

图 18 ping 百度失败

允许所有的 UDP 报文通过, 按照图 19 设置添加规则。因为 DNS 服务是建立在 UDP 协议上的, 因此这时达到的效果是可以使用 DNS 服务, 如图 20, 使用 ping 命令对百度的域名解析成功。

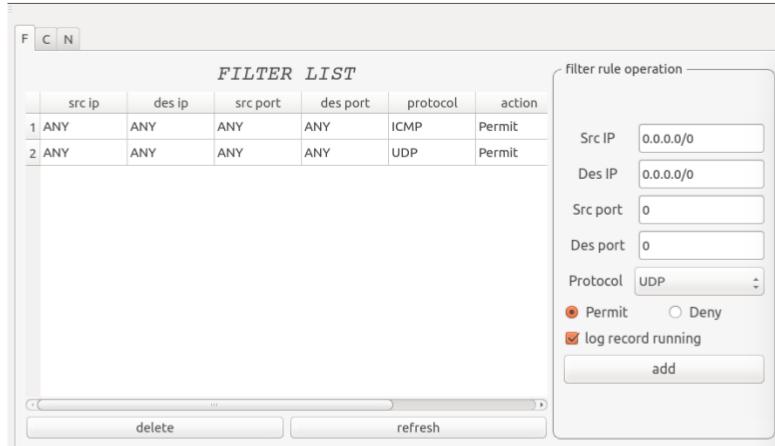


图 19 添加 UDP 的规则

```
root@ubuntu:/home/fhy# ping www.baidu.com
PING www.a.shifen.com (182.61.200.7) 56(84) bytes of data.
64 bytes from 182.61.200.7: icmp_seq=1 ttl=128 time=23.8 ms
64 bytes from 182.61.200.7: icmp_seq=2 ttl=128 time=30.1 ms
64 bytes from 182.61.200.7: icmp_seq=3 ttl=128 time=32.6 ms
^C
--- www.a.shifen.com ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 11031ms
rtt min/avg/max/mdev = 23.854/28.879/32.680/3.705 ms
root@ubuntu:/home/fhy#
```

图 20 解析百度域名成功

防火墙阻止了所有 TCP 包通过，而 ssh 协议是使用 TCP 协议作为其传输层协议的，因此 ssh 服务都不能正常工作。现在允许所有的 TCP 报文通过，按照图 21 设置添加规则。此时能远程 ssh 连接，如图 22。

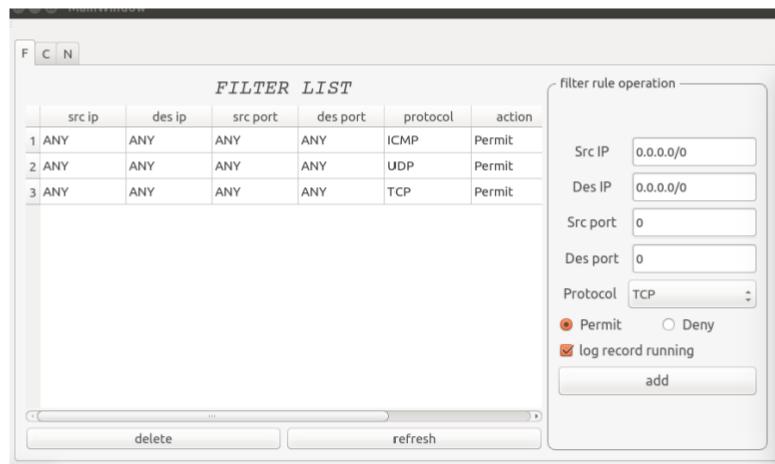


图 21 添加 TCP 规则

```
root@ubuntu:/home/fhy# ssh -p 29286 root@174.137.56.176
The authenticity of host '[174.137.56.176]:29286 ([174.137.56.176]:29286)' can't be established.
RSA key fingerprint is 8c:6a:31:b5:2a:9a:93:35:31:62:4d:c6:a0:95:6a:35.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '[174.137.56.176]:29286' (RSA) to the list of known hosts.
root@174.137.56.176's password: [REDACTED]
```

图 22 远程 ssh 连接

此时点击 Connection State 页，可查看到当前所有连接，如图 23 所示。

	src ip	src port	des ip	des port	protocol	starttime	lifetime
1	127.0.0.1	52874	127.0.1.1	53	UDP	2020.1.6 3:....	9s
2	127.0.0.1	ANY	127.0.1.1	ANY	ICMP	2020.1.6 3:....	9s
3	192.168.6....	39184	174.137.56...	29286	TCP	2020.1.6 3:....	12s
4	192.168.6....	61976	192.168.6.2	53	UDP	2020.1.6 3:....	9s
5	192.168.6....	28441	192.168.6.2	53	UDP	2020.1.6 3:....	9s
6	192.168.6....	22959	192.168.6.2	53	UDP	2020.1.6 3:....	9s
7	127.0.0.1	50037	127.0.1.1	53	UDP	2020.1.6 3:....	9s
8	192.168.6....	58733	192.168.6.2	53	UDP	2020.1.6 3:....	9s

图 23 连接状态表

## 七、心得体会

这次课设的困难主要体现在两个方面：代码的复杂度和调试的困难性上。

这次课设代码的实现主要包括 2 个部分，内核模块的编写以及使用 QT 实现用户 UI 界面。内核模块的编写涉及到 Linux 内核编程，这个在之前的实验和课设中是基本上没有接触过的，所以在编写过程中需要我去进行更多了解和实践。同时，QT 界面的设计也相比之前课设要难上许多，不管是使用到的控件的种类还是数量都要多很多，所以在这个上面就花费了不少时间去熟悉掌握这些控件的使用。

关于程序的调试，这次课设也和以往的不太一样，因为没有直接使用的 debug 调试工具，只能凭借自己的眼睛去看，这就大大的提高了调试的困难程度。刚开始写好一个初步的小框架之后，试着运行起来，然后果然崩掉了，系统直接卡死了，只能重启甚至恢复快照。后来又遇到无数次内核崩掉的问题，这时我才明白老师写在 ppt 最后面的一句“内核指针出错，很容易造成系统崩溃”的分量有多重了。经过艰难的排查，最后发现果真几乎所有的崩溃都是因为内核指针引起的。经过这次课设的教训，也使我编写代码更认真细心了。

印象最深刻的一个 bug 就是程序 open 不了模块，经过不断检索解决途径，最后找到问题的根源竟然是因为权限不足导致的。

通过这次课设中，我接触了解了 Linux 内核编程，提高了我的 Linux 内核编程的能力，对计算机网络安全课堂上讲的知识也有了更深刻的理解。

总之，经过这次课设，我收获颇丰。