原始套接字抓取所有以太网数据包与分析_SoldierJazz2021的博客-CSDN博客

If you have any idea, just send comments to me.

####1.原始套接字介绍
关于socket使用客户机/服务器模型的 SOCK_STREAM 或者 SOCK_DGRAM 用于 TCP 和 UDP 连接的应用更为普遍一些，而如果考虑到从网卡中直接捕获原始报文数据就需要用到原始套接字 SOCK_RAW 类型了。其中原始套接字根据 socket 选项可以工作在网络不同层级上。如果 socket 的第一个参数 domain 设置为 AF_INET 那么套接字就工作在 IP 层，如果设置为 AF_PACKET, 那么套接字就工作在网络接口层和 IP层；本文所给例程将使用后者以便于抓取更多协议类型的数据；关于 socket 最后一个参数 protocol 需要根据第一个参数来选择，本文使用 ETH_P_ALL。更多的使用细节参考 socket 和 protocols 的 man page 即可；

####2.网卡模式
默认情况下网卡只接收 MAC 地址和自己相关的数据包，因此要抓取网络中所有数据包需要将网卡设置为混杂模式，关于混杂模式请参阅我的其他博客。在编程实现上，通过 ioctl 即可将我们程序中设定的参数传递给网卡驱动以实现控制，同样关闭混杂模式也是通过该方法；

####3.数据解析
首先 linux 系统头文件中已经提供好了所有协议类型相关的头文件，这点也可以在例程中发现。但作为程序员，还是要十分清楚每一种协议下报文的基本结构以及报文中每一个字段的含义，关于报文结构也请参阅我的其他博文。在下面解析程序里也可以对每种报文协议略知一二。

####4.源代码
代码如下，具体使用步骤可以阅读代码，也可以直接输入： ./capture -h 来查看用法。关于其中的数据类型最好配合内核源代码进行查看，也更利于协议记忆。另外数据读取采取了最基本的 while 循环解析模式，为了提升效率可以采用 libevent 进行实现。当然目前缺点是退出 while(1) 循环会直接退出程序，无法取消网卡混杂模式和关闭套接字，优化任务就交给你们啦。
PS：如果在网卡上抓取到了大于 MTU 的数据包，不要慌张， 这是正常现象。解决办法参考我的其他博文，或者 send comments to me ? 。

```
/* normal header files */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>
#include <string.h>
#include <signal.h>

/* network header files */
#include <arpa/inet.h>
#include <netdb.h>
#include <linux/if_ether.h>
#include <linux/igmp.h>
#include <netinet/ip_icmp.h>
#include <netinet/in.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
```

```
#include <netinet/udp.h>
#include <net/if.h>
#include <net/ethernet.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/stat.h>
#include <linux/if_arp.h>

/* type definations */
struct global_info{
        unsigned int bytes;
        unsigned int packet_all;

        unsigned int packet_arp;
        unsigned int packet_rarp;

        unsigned int packet_ip;
        unsigned int packet_icmp;
        unsigned int packet_igmp;

        unsigned int packet_tcp;
        unsigned int packet_udp;

        bool print_flag_frame;
        bool print_flag_arp;
        bool print_flag_rarp;
        bool print_flag_ip;
        bool print_flag_icmp;
        bool print_flag_igmp;
        bool print_flag_tcp;
        bool print_flag_udp;
};

struct ip_pair {
        unsigned int source_ip;
        unsigned int dest_ip;
};

/* varibles */
struct global_info global;

struct ip_pair ip_pair[1000];

/* function declaration */
void mac_to_str(char *buf, char *mac_buf);

void init_global(struct global_info *info)
{
        info->bytes = 0;
        info->packet_all = 0;

        info->packet_arp = 0;
        info->packet_rarp = 0;
        info->packet_ip = 0;
        info->packet_icmp = 0;
        info->packet_igmp = 0;
        info->packet_tcp = 0;
        info->packet_udp = 0;

        info->print_flag_arp = false;
        info->print_flag_rarp = false;
        info->print_flag_ip = false;
        info->print_flag_icmp = false;
        info->print_flag_igmp = false;
        info->print_flag_tcp = false;
        info->print_flag_udp = false;
}

void print_global(struct global_info *info)
{
        printf("=============== GLOBAL MESSAGE ===============\n");
        printf("Capture size: %.1f KB\n", (float)(info->bytes / 1024));
        printf("%d packet captured.\n", info->packet_all);
        if (info->packet_arp)
                printf("Num of arp packet: %d\n", info->packet_arp);
        if (info->packet_rarp)
                printf("Num of rarp packet: %d\n", info->packet_rarp);
        if (info->packet_ip)
                printf("Num of ip packet: %d\n", info->packet_ip);
        if (info->packet_icmp)
                printf("Num of icmp packet: %d\n", info->packet_icmp);
        if (info->packet_igmp)
                printf("Num of igmp packet: %d\n", info->packet_igmp);
        if (info->packet_tcp)
                printf("Num of tcp packet: %d\n", info->packet_tcp);
        if (info->packet_udp)
                printf("Num of udp packet: %d\n", info->packet_udp);
        printf("\n");
}

void error_and_exit(char *msg, int code)
{
        herror(msg);
        exit(code);
}

/* excute when interrupted */
void sig_int(int sig)
{
        print_global(&global);
        exit(0);
}

void help(const char *name)
{
        printf("%s: usage: %s [-h][proto1][proto2]...\n", name, name);
        printf("default: print all packet\n");
}

void set_card_promisc(char *intf_name, int sock)
{
        struct ifreq ifr;

        strncpy(ifr.ifr_name, intf_name, strlen(intf_name) + 1);

        if (ioctl(sock, SIOCGIFFLAGS, &ifr) == -1) {
                error_and_exit("ioctl", 2);
        }

        ifr.ifr_flags |= IFF_PROMISC;

        if (ioctl(sock, SIOCSIFFLAGS, &ifr) == -1) {
                error_and_exit("ioctl", 3);
        }
}

void set_card_unpromisc(char *intf_name, int sock)
{
        struct ifreq ifr;
        strncpy(ifr.ifr_name, intf_name, strlen(intf_name) + 1);

        if (ioctl(sock, SIOCGIFFLAGS, &ifr) == -1) {
                error_and_exit("ioctl", 4);
        }

        ifr.ifr_flags &= ~IFF_PROMISC;

        if (ioctl(sock, SIOCSIFFLAGS, &ifr) == -1) {
                error_and_exit("ioctl", 5);
        }
}

void ip_count(struct iphdr *iph)
{
        ip_pair[global.packet_ip - 1].source_ip = iph->saddr;
        ip_pair[global.packet_ip - 1].dest_ip = iph->daddr;
}

void print_icmp(struct icmphdr *picmp)
{
        printf("=============== ICMP PACKET MESSAGE ===============\n");

        printf("Message type:%d\n", picmp->type);
        printf("Suboption: %d\n", picmp->code);

        switch(picmp->type) {
        case ICMP_ECHOREPLY:
                printf("Echo Reply\n");
                break;
        case ICMP_DEST_UNREACH:
                switch (picmp->code) {
                case ICMP_NET_UNREACH:
                        printf("Network Unreachable\n");
                        break;
                case ICMP_HOST_UNREACH:
                        printf("Host Unreachable\n");
                        break;
                case ICMP_PROT_UNREACH:
                        printf("Protocol Unreachable\n");
                        break;
                case ICMP_PORT_UNREACH:
                        printf("Port Unreachable\n");
                        break;
                case ICMP_FRAG_NEEDED:
                        printf("Fragmentation Needed/DF set\n");
                        break;
                case ICMP_SR_FAILED:
                        printf("Source Route failed\n");
                        break;
                case ICMP_NET_UNKNOWN:
                        printf("Network Unknown\n");
                        break;
                case ICMP_HOST_UNKNOWN:
                        printf("Host Unknown\n");
                        break;
                case ICMP_HOST_ISOLATED:
                        printf("Host isolated\n");
                        break;
                case ICMP_NET_ANO:
                        printf("Network Prohibited\n");
                        break;
                case ICMP_HOST_ANO:
                        printf("Host Prohibited\n");
```

```
                                break;
                        case ICMP_NET_UNR_TOS:
                                printf("Network Unreachable cause Service type TOS\n");
                                break;
                        case ICMP_HOST_UNR_TOS:
                                printf("Host Unreachable cause Service type TOS\n");
                                break;
                        case ICMP_PKT_FILTERED:
                                printf("Packet filtered\n");
                                break;
                        case ICMP_PREC_VIOLATION:
                                printf("Precedence violation\n");
                                break;
                        case ICMP_PREC_CUTOFF:
                                printf("Precedence cut off\n");
                                break;
                        default:
                                printf("Code Unknown\n");
                                break;
                        }
                        break;
                case ICMP_SOURCE_QUENCH:
                        printf("Source Quench\n");
                        break;
                case ICMP_REDIRECT:
                        switch( picmp->code ){
                        case ICMP_REDIR_NET:
                                printf("Redirect Net\n");
                                break;
                        case ICMP_REDIR_HOST:
                                printf("Redirect Host\n");
                                break;
                        case ICMP_REDIR_NETTOS:
                                printf("Redirect Net for TOS\n");
                                break;
                        case ICMP_REDIR_HOSTTOS:
                                printf("Redirect Host for TOS\n");
                                break;
                        defalut:
                                printf("Code Unknown\n");
                                break;
                        }
                        break;
                case ICMP_ECHO:
                        printf("Echo Request\n");
                        break;
                case ICMP_TIME_EXCEEDED:
                        switch (picmp->type) {
                        case ICMP_EXC_TTL:
                                printf("TTL count exceeded\n");
                                break;
                        case ICMP_EXC_FRAGTIME:
                                printf("Fragment Reass time exceeded\n");
                                break;
                        default:
                                printf("Code Unknown\n");
                                break;
                        }
                        break;
                case ICMP_PARAMETERPROB:
                        switch (picmp->code) {
                        case 0:
                                printf("IP Header Error\n");
                                break;
                        case 1:
```

```
                        printf("Lack necessary options\n");
                        break;
                default:
                        printf("Reason Unknown\n");
                        break;
                }
                break;
        case ICMP_TIMESTAMP:
                printf("Timestamp Request\n");
                break;
        case ICMP_TIMESTAMPREPLY:
                printf("Timestamp Reply\n");
                break;
        case ICMP_INFO_REQUEST:
                printf("Infomation Request\n");
                break;
        case ICMP_INFO_REPLY:
                printf("Infomation Reply\n");
                break;
        case ICMP_ADDRESS:
                printf("Address Mask Request\n");
                break;
        case ICMP_ADDRESSREPLY:
                printf("Address Mask Reply\n");
                break;
        default:
                printf("Message Type Unknown\n");
                break;
        }
        printf("Checksum: 0x%x\n", ntohs(picmp->checksum));
}

void do_icmp(char *data)
{
    struct icmphdr *picmp = (struct icmphdr *)data;

        global.packet_icmp++;
    if (global.print_flag_icmp)
        print_icmp(picmp);
}

void print_igmp(struct igmphdr *pigmp)
{
        printf("=============== IGMP PACKET MESSAGE ===============\n");
        printf("igmp version: %d\n", pigmp->type & 15);
        printf("igmp type: %d\n", pigmp->type >> 4);
        printf("igmp code: %d\n", pigmp->code);
        printf("igmp checksum: %d\n", ntohs(pigmp->csum));
        printf("igmp group addr: %d\n", ntohl(pigmp->group));
}

void do_igmp(char *data)
{
        struct igmphdr *pigmp = (struct igmphdr *)data;

        global.packet_igmp++;
        if (global.print_flag_igmp)
                print_igmp(pigmp);
}

void print_tcp(struct tcphdr *ptcp, unsigned char ihl, unsigned short itl)
{
        char *data = (char *)ptcp;
        unsigned short tcp_length;
    printf("=============== TCP HEAD MESSAGE ===============\n");
    printf("Source port: %d\n", ntohs(ptcp->source));
    printf("Destination port: %d\n", ntohs(ptcp->dest));
    printf("Seq number: %u\n", ntohl(ptcp->seq));
    printf("Ack number: %u\n", ntohl(ptcp->ack_seq));
    printf("Head Length: %d\n", ptcp->doff * 4);
    printf("6 flags: \n");
    printf("    urg: %d\n", ptcp->urg);
    printf("    ack: %d\n", ptcp->ack);
    printf("    psh: %d\n", ptcp->psh);
    printf("    rst: %d\n", ptcp->rst);
    printf("    syn: %d\n", ptcp->syn);
    printf("    fin: %d\n", ptcp->fin);
    printf("Window size (16bits): %d\n", ntohs(ptcp->window));
    printf("Checksum (16bits): %d\n", ntohs(ptcp->check));
    printf("Urg (16bits): %d\n", ntohs(ptcp->urg_ptr));

    if (ptcp->doff * 4 == 20) {
        printf("Option Data: None\n");
    } else {
        printf("Option Data: %d bytes\n", ptcp->doff * 4 - 20);
    }
        tcp_length = itl - ihl - ptcp->doff * 4;
    data += ptcp->doff * 4;
    printf("TCP Data length: %d bytes\n", tcp_length);
        if (tcp_length < 2000) {
                for (int i = 1; i < tcp_length; i++)
                        printf("TCP Data: 0x%02x\n", (unsigned char)(*data++));
        }
        printf("\n");
}

void do_tcp(char *data, unsigned char ihl, unsigned short itl)
{
        struct tcphdr *ptcp;

        global.packet_tcp++;
        ptcp = (struct tcphdr *)data;
        if (global.print_flag_tcp)
                print_tcp(ptcp, ihl, itl);
}

void print_udp(struct udphdr *pudp)
{
        char *data;
        unsigned short udp_length;

        printf("========== UDP PACKET MESSAGE ==========\n");
        printf("Source Port (16 bits): %d\n", ntohs(pudp->source));
        printf("Destination Port (16 bits): %d\n", ntohs(pudp->dest));
        printf("UDP Length (16 bits): %d\n", ntohs(pudp->len));
        printf("UDP Checksum (16 bits): %d\n", ntohs(pudp->check));

        udp_length = ntohs(pudp->len) - sizeof(struct udphdr);
    printf("UDP Data length: %d bytes\n", udp_length);

        if (udp_length) {
                data = (char *)pudp + sizeof(struct udphdr);
                for (int i = 1; i < udp_length; i++)
                        printf("UDP Data: 0x%02x\n", (unsigned char)(*data++));
        }
        printf("\n");
}

void do_udp(char *data)
```

```
{
        struct udphdr *pudp = (struct udphdr *)data;

        global.packet_udp++;
        if (global.print_flag_udp)
                print_udp(pudp);
}

void print_ip(struct iphdr *iph)
{
        printf("=============== IP HEAD MESSAGE ===============\n");
        printf("IP head length: %d\n", iph->ihl * 4);
        printf("IP version: %d\n", iph->version);
        printf("Service type (tos): %d\n", iph->tos);
        printf("Data packet length: %d\n", ntohs(iph->tot_len));
        printf("ID(16 bits): %d\n", ntohs(iph->id));
        printf("Frag off(16 bits): %d\n", ntohs(iph->frag_off));
        printf("Survival time(8 bits): %d\n", iph->ttl);
        printf("IP protocol: %d\n", iph->protocol);
        printf("Checksum: 0x%4x\n", ntohs(iph->check));
        printf("Source IP addr(32 bits): %s\n", inet_ntoa(*(struct in_addr *)(&iph->saddr)));
        printf("Destination IP addr(32 bits): %s\n", inet_ntoa(*(struct in_addr *)(&iph->daddr)));
        printf("\n");
}

void do_ip(char *data)
{
        struct iphdr *pip = (struct iphdr *)data;

        /* 4 bits of ip head length, 1 stand 32bit data */
        unsigned char ip_head_length = pip->ihl * 4;
        unsigned short ip_total_length = ntohs(pip->tot_len);
        char *pdata = data + ip_head_length;

        global.packet_ip++;
        if (global.print_flag_ip)
                print_ip(pip);

        ip_count(pip);

        switch (pip->protocol) {
        case IPPROTO_ICMP:
                do_icmp(pdata);
                break;
        case IPPROTO_IGMP:
                do_igmp(pdata);
                break;
        case IPPROTO_TCP:
                do_tcp(pdata, ip_head_length, ip_total_length);
                break;
        case IPPROTO_UDP:
                do_udp(pdata);
                break;
        default:
                printf("Unknown IP type: 0x%2x", pip->protocol);
                break;
        }
}

void print_arp( struct arphdr * parp )
{
        char *addr = (char*)(parp + 1);
        char buf[18];

        printf("Hardware Type: (%d) ", ntohs(parp->ar_hrd));
```

```
switch (ntohs(parp->ar_hrd)) {
case ARPHRD_ETHER:
        printf("Ethernet 10Mbps.\n");
        break;
case ARPHRD_EETHER:
        printf("Experimental Ethernet.\n");
        break;
case ARPHRD_AX25:
        printf("AX.25 Level 2.\n");
        break;
case ARPHRD_PRONET:
        printf("PROnet token ring.\n");
        break;
case ARPHRD_IEEE802:
        printf("IEEE 802.2 Ethernet/TR/TB.\n");
        break;
case ARPHRD_APPLETLK:
        printf("APPLEtalk.\n");
        break;
case ARPHRD_ATM:
        printf("ATM.\n");
        break;
case ARPHRD_IEEE1394:
        printf("IEEE 1394 IPv4 - RFC 2734.\n");
        break;
default:
        printf("Unknown Hardware Type.\n");
        break;
}
printf("Protocol Type: (%d)", ntohs(parp->ar_pro));
switch (ntohs(parp->ar_pro)) {
case ETHERTYPE_IP:
        printf("IP.\n");
        break;
default:
        printf("error.\n");
        break;
}
printf("Hardware addr length: %d\n", parp->ar_hln);
printf("Protocol addr length: %d\n", parp->ar_pln);
printf("ARP opcode(command): %d\n", ntohs(parp->ar_op));
switch (ntohs(parp->ar_op)) {
case ARPOP_REQUEST:
        printf("ARP request.\n");
        break;
case ARPOP_REPLY:
        printf("ARP reply.\n");
        break;
case ARPOP_RREQUEST:
        printf("RARP request.\n");
        break;
case ARPOP_RREPLY:
        printf("RARP reply.\n");
        break;
case ARPOP_InREQUEST:
        printf("InARP request.\n");
        break;
case ARPOP_InREPLY:
        printf("InARP reply.\n");
        break;
case ARPOP_NAK:
        printf("(ATM)ARP NAK.\n");
        break;
default:
        printf("Unknown ARP opcode.\n");
```

```
                break;
        }

        mac_to_str(buf, addr);
        printf("The Source MAC addr: %s\n", buf );
        printf("The Source IP addr: %s\n", inet_ntoa(*(struct in_addr *)(addr+6)));
        mac_to_str(buf, addr + 10);
        printf("The Destination MAC addr: %s\n", buf );
        printf("The Destination IP addr: %s\n", inet_ntoa(*(struct in_addr *)(addr+16)));
}

void do_arp(char *data)
{
        struct arphdr *parp;

        global.packet_arp++;
        parp = (struct arphdr *)data;
        if (global.print_flag_arp) {
                printf("========== ARP PACKET MESSAGE ==========\n");
                print_arp(parp);
        }
}

void do_rarp(char *data)
{
        struct arphdr *parp = (struct arphdr *)data;

        global.packet_rarp++;
        if (global.print_flag_rarp) {
                printf("========== ARP PACKET MESSAGE ==========\n");
                print_arp(parp);
        }
}

void mac_to_str(char *buf, char *mac_buf)
{
        sprintf(buf, "%02x:%02x:%02x:%02x:%02x:%02x\n", (unsigned char)*mac_buf,
                        (unsigned char)(*(mac_buf + 1)), (unsigned char)(*(mac_buf + 2)),
                        (unsigned char)*(mac_buf + 3), (unsigned char)(*(mac_buf + 4)),
                        (unsigned char)*(mac_buf + 5));
        buf[17] = 0;
}

void print_frame(struct ether_header *peth)
{
        char buf[18];
        char *dhost;
        char *shost;

        printf("=============== ETHERNET MESSAGE IN PACKET %d ===============\n", global.packet_all);
        dhost = peth->ether_dhost;
        mac_to_str(buf, dhost);
        printf("The Destination MAC addr: %s\n", buf);
        shost = peth->ether_shost;
        mac_to_str(buf, shost);
        printf("The Source MAC addr: %s\n", buf);
        printf("\n");
}

void do_frame(int sock)
{
        char frame_buf[2000];
        int recv_num;
        struct sockaddr src_addr;
        int addrlen;
        struct ether_header *peth;
        char *pdata;

        addrlen = sizeof(struct sockaddr);
        bzero(frame_buf, sizeof(frame_buf));
        recv_num = recvfrom(sock, frame_buf, sizeof(frame_buf), 0, &src_addr, &addrlen);

        global.packet_all++;
        global.bytes += recv_num;

        peth = (struct ether_header *)frame_buf;

        if (global.print_flag_frame)
                print_frame(peth);

        pdata = frame_buf + sizeof(struct ether_header);

        switch(ntohs(peth->ether_type)) {
        case ETHERTYPE_PUP:
                break;
        case ETHERTYPE_IP:
                do_ip(pdata);
                break;
        case ETHERTYPE_ARP:
                do_arp(pdata);
                break;
        case ETHERTYPE_REVARP:
                do_rarp(pdata);
                break;
        default:
                printf("Unknown ethernet type 0x%x(%d).\n", ntohs(peth->ether_type),
                                ntohs(peth->ether_type));
        }

}

int main(int argc, const char *argv[])
{
        int sock_fd;

        init_global(&global);

        if (argc == 1) {
                global.print_flag_frame = true;
                global.print_flag_arp = true;
                global.print_flag_rarp = true;
                global.print_flag_ip = true;
                global.print_flag_icmp = true;
                global.print_flag_igmp = true;
                global.print_flag_tcp = true;
                global.print_flag_udp = true;
        } else {
                if (!strcasecmp(argv[1], "-h")) {
                        help(argv[0]);
                        exit(0);
                }
                else {
                        int i = 1;
                        for (i = 1; i < argc; i++) {
                                if (!strcasecmp(argv[i], "frame"))
                                        global.print_flag_frame = true;
                                else if (!strcasecmp(argv[i], "arp"))
                                        global.print_flag_arp = true;
                                else if (!strcasecmp(argv[i], "rarp"))
                                        global.print_flag_rarp = true;
                                else if (!strcasecmp(argv[i], "ip"))
                                        global.print_flag_ip = true;
                                else if (!strcasecmp(argv[i], "icmp"))
                                        global.print_flag_icmp = true;
                                else if (!strcasecmp(argv[i], "igmp"))
                                        global.print_flag_igmp = true;
                                else if (!strcasecmp(argv[i], "tcp"))
                                        global.print_flag_tcp = true;
                                else if (!strcasecmp(argv[i], "udp"))
                                        global.print_flag_udp = true;
                                else
                                        error_and_exit("error protocol arg", 1);
                        }
                }
        }

        if ((sock_fd = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL))) == -1)
                error_and_exit("socket", 1);

        signal(SIGINT, sig_int);

        set_card_promisc("ens33", sock_fd);

        while(1) {
                do_frame(sock_fd);
        }

        set_card_unpromisc("ens33", sock_fd);
        close(sock_fd);

        return 0;
}
```

- 225
- 226
- 227
- 228
- 229
- 230
- 231
- 232
- 233
- 234
- 235
- 236
- 237
- 238
- 239
- 240
- 241
- 242
- 243
- 244
- 245
- 246
- 247
- 248
- 249
- 250
- 251
- 252
- 253
- 254
- 255
- 256
- 257
- 258
- 259
- 260
- 261
- 262
- 263
- 264
- 265
- 266
- 267
- 268
- 269
- 270
- 271
- 272
- 273
- 274
- 275
- 276
- 277
- 278
- 279
- 280
- 281
- 282
- 283
- 284
- 285
- 286
- 287
- 288
- 289

- 290
- 291
- 292
- 293
- 294
- 295
- 296
- 297
- 298
- 299
- 300
- 301
- 302
- 303
- 304
- 305
- 306
- 307
- 308
- 309
- 310
- 311
- 312
- 313
- 314
- 315
- 316
- 317
- 318
- 319
- 320
- 321
- 322
- 323
- 324
- 325
- 326
- 327
- 328
- 329
- 330
- 331
- 332
- 333
- 334
- 335
- 336
- 337
- 338
- 339
- 340
- 341
- 342
- 343
- 344
- 345
- 346
- 347
- 348
- 349
- 350
- 351
- 352
- 353
- 354

- 420
- 421
- 422
- 423
- 424
- 425
- 426
- 427
- 428
- 429
- 430
- 431
- 432
- 433
- 434
- 435
- 436
- 437
- 438
- 439
- 440
- 441
- 442
- 443
- 444
- 445
- 446
- 447
- 448
- 449
- 450
- 451
- 452
- 453
- 454
- 455
- 456
- 457
- 458
- 459
- 460
- 461
- 462
- 463
- 464
- 465
- 466
- 467
- 468
- 469
- 470
- 471
- 472
- 473
- 474
- 475
- 476
- 477
- 478
- 479
- 480
- 481
- 482
- 483
- 484

- 485
- 486
- 487
- 488
- 489
- 490
- 491
- 492
- 493
- 494
- 495
- 496
- 497
- 498
- 499
- 500
- 501
- 502
- 503
- 504
- 505
- 506
- 507
- 508
- 509
- 510
- 511
- 512
- 513
- 514
- 515
- 516
- 517
- 518
- 519
- 520
- 521
- 522
- 523
- 524
- 525
- 526
- 527
- 528
- 529
- 530
- 531
- 532
- 533
- 534
- 535
- 536
- 537
- 538
- 539
- 540
- 541
- 542
- 543
- 544
- 545
- 546
- 547
- 548
- 549

- 550
- 551
- 552
- 553
- 554
- 555
- 556
- 557
- 558
- 559
- 560
- 561
- 562
- 563
- 564
- 565
- 566
- 567
- 568
- 569
- 570
- 571
- 572
- 573
- 574
- 575
- 576
- 577
- 578
- 579
- 580
- 581
- 582
- 583
- 584
- 585
- 586
- 587
- 588
- 589
- 590
- 591
- 592
- 593
- 594
- 595
- 596
- 597
- 598
- 599
- 600
- 601
- 602
- 603
- 604
- 605
- 606
- 607
- 608
- 609
- 610
- 611
- 612
- 613
- 614

- 615
- 616
- 617
- 618
- 619
- 620
- 621
- 622
- 623
- 624
- 625
- 626
- 627
- 628
- 629
- 630
- 631
- 632
- 633
- 634
- 635
- 636
- 637
- 638
- 639
- 640
- 641
- 642
- 643
- 644
- 645
- 646
- 647
- 648
- 649
- 650
- 651
- 652
- 653
- 654
- 655
- 656
- 657
- 658
- 659
- 660
- 661
- 662
- 663
- 664
- 665
- 666
- 667
- 668
- 669
- 670
- 671
- 672
- 673
- 674
- 675
- 676
- 677
- 678
- 679

- 680
- 681
- 682
- 683
- 684
- 685
- 686
- 687
- 688
- 689
- 690
- 691
- 692
- 693
- 694
- 695
- 696
- 697
- 698
- 699
- 700
- 701
- 702