网 络 工 程
本 科 实 验 报 告

实验名称： SDN 可编程网络实验

学 员 姓 名	程景愉	学 号	202302723005
指 导 教 员	胡罡	职 称	教授
实 验 室	306-707	实 验 时 间	2025.11.18

国防科技大学教育训练部制

《本科实验报告》填写说明

实验报告内容编排应符合以下要求：

（1）采用 A4（21cm×29.7cm）白色复印纸，单面黑字。上下左右各侧的页边距均为 3cm；缺省文档网格：字号为小 4 号，中文为宋体，英文和阿拉伯数字为 Times New Roman，每页 30 行，每行 36 字；页脚距边界为 2.5cm，页码置于页脚、居中，采用小 5 号阿拉伯数字从 1 开始连续编排，封面不编页码。

（2）报告正文最多可设四级标题，字体均为黑体，第一级标题字号为 4 号，其余各级标题为小 4 号；标题序号第一级用“一、”、“二、”……，第二级用“（一）”、“（二）”……，第三级用“1.”、“2.”……，第四级用“（1）”、“（2）”……，分别按序连续编排。

（3）正文插图、表格中的文字字号均为 5 号。

目录

1 实验概述 5
1.1 实验内容 5
1.2 实验要求 5
2 编写 SDN 交换机源码处理 OpenFlow 协议消息 5
2.1 本次实验所编写的协议 5
2.1.1 实现的协议 5
2.2 相关数据结构 6
2.3 消息头构建 6
2.4 消息处理函数 6
2.4.1 处理 Hello 消息 6
2.4.2 处理 Features Request 消息 6
2.4.3 处理 Get Config Request 消息 7
2.4.4 处理 Description Request 消息 7
2.4.5 处理 Flow Stats Request 消息 8
2.4.6 处理 Table Stats Request 消息 9
2.4.7 处理 Port Stats Request 消息 9
2.4.8 处理 Group Features Request 消息 10
2.4.9 处理 Port Description Request 消息 10
2.4.10 处理 Packet Out 消息 11
2.4.11 处理 Role Request 消息 12
2.5 实验结果 13
2.5.1 OFPT_ROLE_REPLY 13
2.5.2 OFPT_MULTIPART_REPLY 14
3 实验总结 16
4 选做实验介绍 17
4.1 实验原理 17
4.1.1 发起 DDoS 攻击 17
4.1.2 防御 DDoS 攻击 18
4.2 选做实验总结 20
参考文献 21

图目录

Figure 1 OFPT_ROLE_REQUEST 消息 13
Figure 2 OFPT_ROLE_REPLY 消息 14
Figure 3 OFPMP_DESC 消息 14
Figure 4 OFPMP_TABLE 消息 15
Figure 5 遭受 DDoS 攻击 17
Figure 6 流表项 18
Figure 7 丢弃异常流量 19
Figure 8 流表项 20

1 实验概述

1.1 实验内容

本次综合实验中分有三个实验：

1. 基于 SDN 交换机源码处理 openflow 协议消息
2. 虚拟网络拓扑实验
3. DDoS 网络攻击的 SDN 实验

本实验报告重点分析第一个实验的实现细节，并在最后补充介绍本小组的选做实验。

1.2 实验要求

1. 熟悉 Openbox-S4 的基本功能
2. 熟悉 OpenFlow 协议的基本消息格式
3. 熟悉 OpenFlow 协议的基本消息处理流程

2 编写 SDN 交换机源码处理 OpenFlow 协议消息

本次实验在`main_user_openflow.c`文件中实现了 OpenFlow 协议处理逻辑。代码通过多个函数处理不同类型的 OpenFlow 消息，包括 Hello 消息、Features Request 消息、Flow Stats Request 消息等。每个函数都根据消息类型生成相应的回复消息，并通过`send_openflow_message`函数发送给控制器。以下将逐一分析每个函数的实现细节。

2.1 本次实验所编写的协议

本次实验实现了 OpenFlow 协议中的大部分消息处理功能。实验代码涵盖了从交换机特性查询到流表统计、端口统计等多种消息类型的处理。以下是实验编写协议的总结。

2.1.1 实现的协议

- OFPT_FEATURES_REQUEST：用于获取交换机支持的流表数量、缓冲区大小等特性信息。对应的处理函数为`handle_opfmsg_features_request`。
- OFPT_GET_CONFIG_REQUEST：用于查询交换机的配置信息，如 Miss Send Length 等。对应的处理函数为`handle_ofpmsg_get_config_request`。
- OFPT_MULTIPART_REQUEST：用于处理多种统计信息请求，包括交换机描述信息、流表信息、端口统计信息等。对应的处理函数包括：
  - `handle_ofpmsg_desc`
  - `handle_ofpmsg_table`
  - `handle_ofpmsg_port_stats`
  - `handle_ofpmsg_group_features`
  - `handle_ofpmsg_port_desc`
- OFPT_PACKET_OUT：用于处理控制器发送的数据包，并根据动作指示将数据包从指定端口发送出去。对应的处理函数为`handle_ofpmsg_packet_out`。
- OFPT_ROLE_REQUEST：用于配置交换机的角色（如主控制器、从控制器等）。对应的处理函数为`handle__opfmsg_role_request`。

2.2 相关数据结构

在 `main_user_openflow.c` 中，定义了一些关键的数据结构和全局变量，用于处理 OpenFlow 协议的消息。主要的数据结构包括 `ofp_header`、`ofp_switch_features`、`ofp_flow_stats` 和 `ofp_port_stats` 等。这些数据结构用于处理 OpenFlow 协议中的不同消息类型，如交换机特性请求、流表统计请求、端口统计请求等。

`ofp_header` 结构体用于表示 OpenFlow 消息头，包含协议版本、消息类型、消息长度和事务 ID 等字段。`ofp_switch_features` 结构体用于表示交换机的特性信息，如数据路径 ID、缓冲区数量、流表数量等。`ofp_flow_stats` 结构体用于表示流表统计信息，如流表项长度、优先级、数据包计数等。`ofp_port_stats` 结构体用于表示端口统计信息，如端口号、接收数据包数、发送数据包数等。

2.3 消息头构建

在 OpenFlow 协议中，每个消息都有一个消息头，用于标识消息的版本、类型、长度和事务 ID。代码中实现了 `build_opfmsg_header` 函数，用于构建 OpenFlow 消息头。该函数接收消息长度、消息类型和事务 ID 作为参数，并填充消息头的各个字段。通过调用该函数，可以确保消息能够被正确解析和处理。

2.4 消息处理函数

代码中实现了多个 OpenFlow 消息处理函数，每个函数对应一种 OpenFlow 消息类型。以下是每个函数的详细分析。

2.4.1 处理 Hello 消息

`handle_opfmsg_hello` 函数用于处理控制器发送的 Hello 消息。Hello 消息是 OpenFlow 协议中的基础消息，用于交换控制器和交换机之间的协议版本信息。函数首先检查消息头中的 `version` 字段，判断协议版本是否为 1.3。如果版本匹配，则打印接收到的 Hello 消息；否则，生成一个错误消息并发送给控制器。函数返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_opfmsg_hello(struct ofp_buffer *ofpbuf) {
    if (ofpbuf->header.version == 0x04) {
        printf("RECV HELLO!\n\n\n");
    } else {
        struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_ERROR, ofpbuf->header.xid, sizeof(struct
ofp_header));
        send_openflow_message(ofpbuf_reply, sizeof(struct ofp_header));
    }
    return HANDLE;
}
```

2.4.2 处理 Features Request 消息

`handle_opfmsg_features_request` 函数用于处理控制器发送的 Features Request 消息。该消息用于查询交换机的特性信息，如支持的流表数量、缓冲区大小等。函数生成一个 Features

Reply 消息，并填充交换机的特性信息，如数据路径 ID、缓冲区数量、流表数量等。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_opfmsg_features_request(struct ofp_buffer *ofpbuf)
{
    int feature_reply_len = sizeof(struct ofp_switch_features) +
sizeof(struct ofp_header);
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_opfmsg_reply_ofpbuf(OFPT_FEATURES_REPLY, ofpbuf->header.xid,
feature_reply_len);
    struct ofp_switch_features *feature_reply_msg = (struct
ofp_switch_features *)ofpbuf_reply->data;

    feature_reply_msg->datapath_id = 0x0100000000000000;
    feature_reply_msg->n_buffers = htonl(46);
    feature_reply_msg->n_tables = 3;
    feature_reply_msg->capabilities = 0x7;

    send_openflow_message(ofpbuf_reply, feature_reply_len);
    return HANDLE;
}
```

2.4.3 处理 Get Config Request 消息

`handle_ofpmsg_get_config_request` 函数用于处理控制器发送的 Get Config Request 消息。该消息用于查询交换机的配置信息，如 Miss Send Length 等。函数生成一个 Get Config Reply 消息，并填充交换机的配置信息。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_get_config_request(struct ofp_buffer
*ofpbuf) {
    int reply_len = sizeof(struct ofp_switch_config) + sizeof(struct
ofp_header);
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_GET_CONFIG_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_switch_config *switch_config_reply = (struct
ofp_switch_config *)ofpbuf_reply->data;

    switch_config_reply->flags = htons(0x0000);
    switch_config_reply->miss_send_len = htons(32);

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.4 处理 Description Request 消息

`handle_ofpmsg_desc` 函数用于处理控制器发送的 Description Request 消息。该消息用于查询交换机的描述信息，如制造商、硬件版本等。函数生成一个 Multipart Reply 消息，并填充交换机的描述信息。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_desc(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct ofp_multipart)
+ sizeof(struct ofp_desc_stats);
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_multipart *ofpmp_reply = (struct ofp_multipart
*)ofpbuf_reply->data;

    static const char *default_mfr_desc = "Wanglixuan";
    static const char *default_hw_desc = "Lixuan_OpenBox";
    static const char *default_sw_desc = "Lixuan_Driver";
    static const char *default_serial_desc = "Lixuan OpenBox Series";
    static const char *default_dp_desc = "None";

    ofpmp_reply->type = htons(OFPMP_DESC);
    ofpmp_reply->flags = htonl(OFPMP_REPLY_MORE_NO);
    snprintf(ofpmp_reply->ofpmp_desc[0].mfr_desc, sizeof ofpmp_reply-
>ofpmp_desc[0].mfr_desc, "%s", default_mfr_desc);
    snprintf(ofpmp_reply->ofpmp_desc[0].hw_desc, sizeof ofpmp_reply-
>ofpmp_desc[0].hw_desc, "%s", default_hw_desc);
    snprintf(ofpmp_reply->ofpmp_desc[0].sw_desc, sizeof ofpmp_reply-
>ofpmp_desc[0].sw_desc, "%s", default_sw_desc);
    snprintf(ofpmp_reply->ofpmp_desc[0].serial_num, sizeof ofpmp_reply-
>ofpmp_desc[0].serial_num, "%s", default_serial_desc);
    snprintf(ofpmp_reply->ofpmp_desc[0].dp_desc, sizeof ofpmp_reply-
>ofpmp_desc[0].dp_desc, "%s", default_dp_desc);

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.5 处理 Flow Stats Request 消息

`handle_ofpmsg_flow_stats` 函数用于处理控制器发送的 Flow Stats Request 消息。该消息用于查询交换机的流表统计信息。函数生成一个 Multipart Reply 消息，并设置消息类型为 Flow Stats。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_flow_stats(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct
ofp_multipart);
    struct ofp_buffer *reply_buffer = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_multipart *multipart_reply = (struct ofp_multipart
*)reply_buffer->data;

    multipart_reply->type = htons(OFPMP_FLOW);
    multipart_reply->flags = htonl(OFPMP_REPLY_MORE_NO);

    send_openflow_message(reply_buffer, reply_len);
    return HANDLE;
}
```

2.4.6 处理 Table Stats Request 消息

`handle_ofpmsg_table` 函数用于处理控制器发送的 Table Stats Request 消息。该消息用于查询交换机的流表统计信息。函数生成一个 Multipart Reply 消息，并填充流表统计信息，如匹配计数、查找计数等。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_table(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct ofp_multipart)
+ sizeof(struct ofp_table_stats) * 1;
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_multipart *ofpmp_reply = (struct ofp_multipart
*)ofpbuf_reply->data;

    ofpmp_reply->type = htons(OFPMP_TABLE);
    ofpmp_reply->flags = htonl(OFPMP_REPLY_MORE_NO);

    ofpmp_reply->table_stats[0].matched_count = htonll(2025);
    ofpmp_reply->table_stats[0].table_id = 0;
    ofpmp_reply->table_stats[0].lookup_count = htonll(46);
    ofpmp_reply->table_stats[0].active_count = htonl(1);

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.7 处理 Port Stats Request 消息

`handle_ofpmsg_port_stats` 函数用于处理控制器发送的 Port Stats Request 消息。该消息用于查询交换机的端口统计信息。函数生成一个 Multipart Reply 消息，并填充端口统计信息，如持续时间、接收数据包数等。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_port_stats(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct ofp_multipart)
+ sizeof(struct ofp_port_stats) * nmps.cnt;
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_multipart *ofpmp_reply = (struct ofp_multipart
*)ofpbuf_reply->data;

    ofpmp_reply->type = htons(OFPMP_PORT_STATS);
    ofpmp_reply->flags = htonl(OFPMP_REPLY_MORE_NO);

    for (int i = 0; i < nmps.cnt; i++) {
        ofpmp_reply->ofpmp_port_stats[i] = nmps.ports[i].stats;
        ofpmp_reply->ofpmp_port_stats[i].duration_sec = htonl(2025);
        ofpmp_reply->ofpmp_port_stats[i].duration_nsec = htonl(51);
    }

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.8 处理 Group Features Request 消息

`handle_ofpmsg_group_features` 函数用于处理控制器发送的 Group Features Request 消息。该消息用于查询交换机的组表特性信息。函数生成一个 Multipart Reply 消息，并填充组表特性信息，如最大组数等。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_group_features(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct
ofp_group_features) + 8;
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_group_features *group = (struct ofp_group_features
*)ofpbuf_reply->data;

    group->types = htons(OFPMP_GROUP_FEATURES);
    group->max_groups[0] = htonl(0xdeadbeef);

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.9 处理 Port Description Request 消息

`handle_ofpmsg_port_desc` 函数用于处理控制器发送的 Port Description Request 消息。该消息用于查询交换机的端口描述信息。函数生成一个 Multipart Reply 消息，并填充端口描述信息，如端口状态等。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_port_desc(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct ofp_multipart)
+ sizeof(struct ofp_port) * nmps.cnt;
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_MULTIPART_REPLY, ofpbuf->header.xid, reply_len);
    struct ofp_multipart *ofpmp_reply = (struct ofp_multipart
*)ofpbuf_reply->data;

    ofpmp_reply->type = htons(OFPMP_PORT_DESC);
    ofpmp_reply->flags = htonl(OFPMP_REPLY_MORE_NO);

    for (int i = 0; i < nmps.cnt; i++) {
        ofpmp_reply->ofpmp_port_desc[i] = nmps.ports[i].state;
    }

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.4.10 处理 Packet Out 消息

`handle_ofpmsg_packet_out` 函数用于处理控制器发送的 Packet Out 消息。该消息用于指示交换机将数据包从指定端口发送出去。函数解析消息中的动作列表，判断动作类型是否为 `OFPAT_OUTPUT`，并根据动作指示的端口号调用 `nms_exec_action` 函数将数据包发送出去。函数返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_ofpmsg_packet_out(struct ofp_buffer *ofpbuf) {
    struct ofp_packet_out *out = (struct ofp_packet_out *)ofpbuf;
    struct ofp_action_output *action = (struct ofp_action_output *)&out-
>actions[0];
    int action_len = ntohs(out->actions_len);
    struct eth_header *eth = (struct eth_header *)&ofpbuf-
>data[sizeof(struct ofp_packet_out) - sizeof(struct ofp_header) +
action_len];
    int send_len = ntohs(ofpbuf->header.length) - sizeof(struct
ofp_packet_out) - action_len;

    if (action_len == 0) {
        nms_exec_action(ntohl(out->in_port), OFPP_FLOOD, eth, send_len, -1);
    } else {
        while (action_len > 0) {
            if (action->type == OFPAT_OUTPUT) {
                nms_exec_action(ntohl(out->in_port), ntohl(action->port),
eth, send_len, -1);
            }
            action_len -= sizeof(struct ofp_action_output);
            action++;
        }
    }
    return HANDLE;
}
```

2.4.11 处理 Role Request 消息

`handle_opfmsg_role_request` 函数用于处理控制器发送的 Role Request 消息。该消息用于配置交换机的角色（如主控制器、从控制器等）。函数生成一个 Role Reply 消息，并填充角色配置信息。通过 `send_openflow_message` 函数将回复消息发送给控制器，并返回 `HANDLE` 表示消息已处理。

```
static enum ofperr handle_opfmsg_role_request(struct ofp_buffer *ofpbuf) {
    int reply_len = sizeof(struct ofp_header) + sizeof(struct ofp_role);
    struct ofp_buffer *ofpbuf_reply = (struct ofp_buffer
*)build_reply_ofpbuf(OFPT_ROLE_REPLY, ofpbuf->header.xid, reply_len);
    memcpy(ofpbuf_reply->data, ofpbuf->data, sizeof(struct ofp_role));
    ofpbuf_reply->header.type = OFPT_ROLE_REPLY;

    send_openflow_message(ofpbuf_reply, reply_len);
    return HANDLE;
}
```

2.5 实验结果

在控制器与交换机之间进行抓包，分析报文的交互过程。控制器发送不同类型的OpenFlow消息给交换机，交换机接收并处理消息，并返回相应的回复消息。通过抓包分析，可以看到消息的交互过程，包括消息头、消息类型、消息长度等字段的解析和处理。用这种方式来验证协议编写的正确性。

实现的子协议种类较多，这里只展示部分报文内容。

2.5.1 OFPT_ROLE_REPLY

OFPT_ROLE_REPLY 类型消息。此消息用于配置交换机的角色（如主控制器、从控制器等）。下面的 Figure 1 是控制器发送的 Role Request 消息。

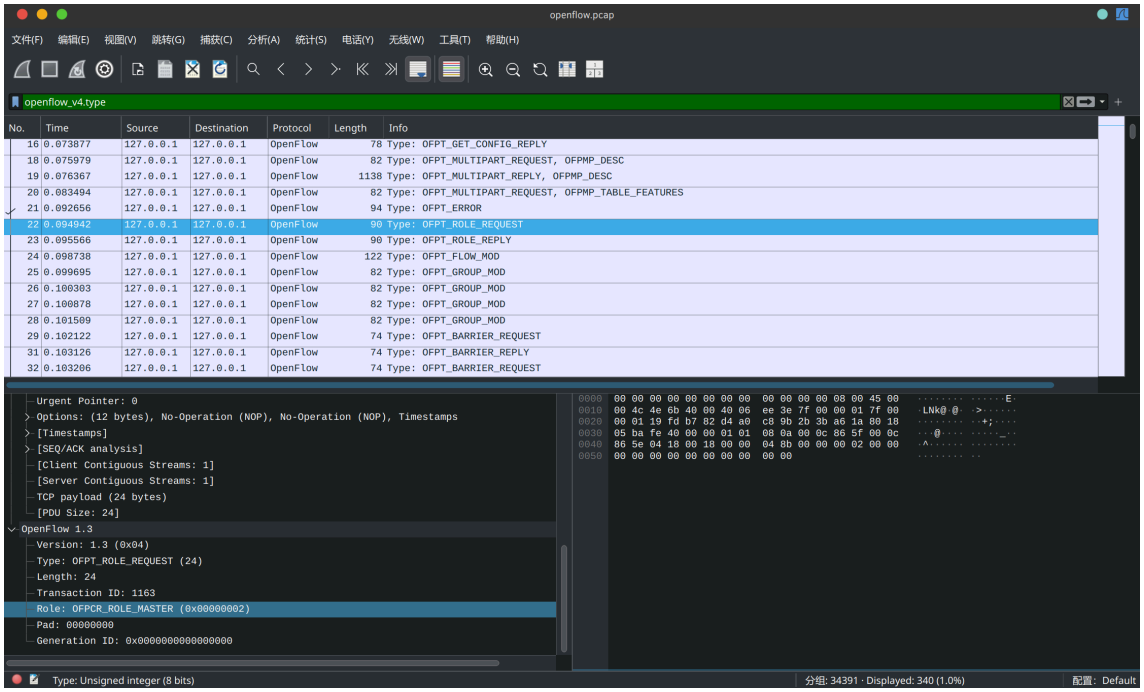

Figure 1: OFPT_ROLE_REQUEST 消息

如图 Figure 2 所示，交换机接收并处理后返回的 Role Reply 消息。

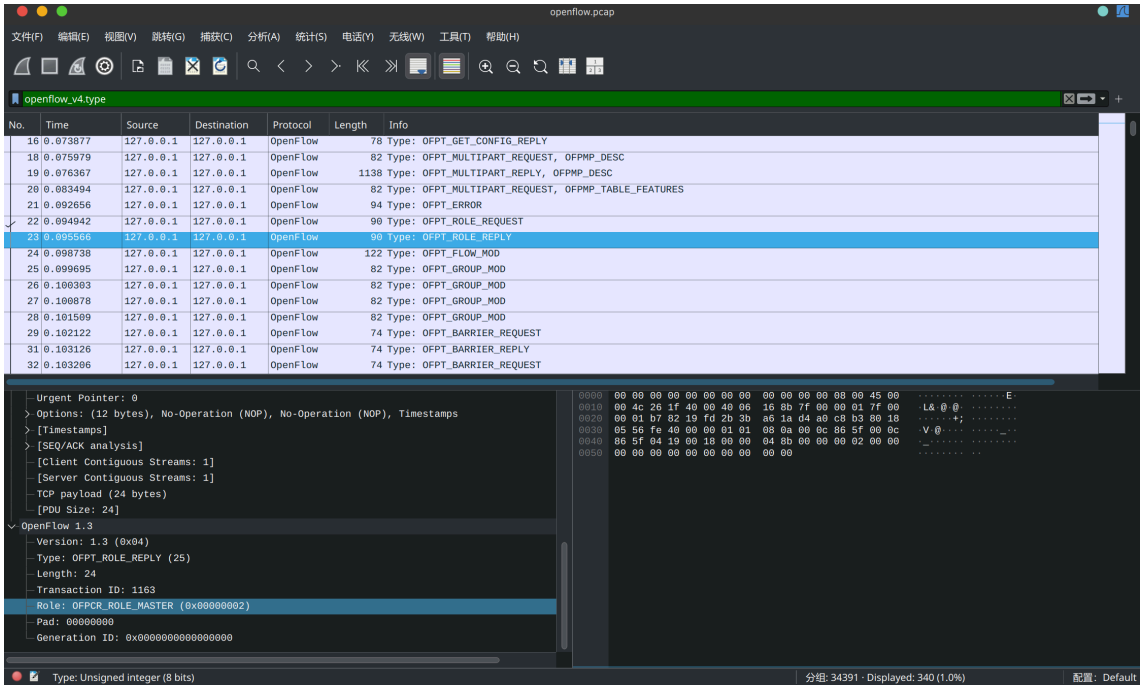

Figure 2: OFPT_ROLE_REPLY 消息

这说明交换机已经接收到了控制器发送的 Role Request 消息，并返回了 Role Reply 消息。

2.5.2 OFPT_MULTIPART_REPLY

OFPMP_DESC 类型消息。此消息用于查询交换机的描述信息，包括制造商、硬件版本、软件版本、序列号和数据路径描述等。

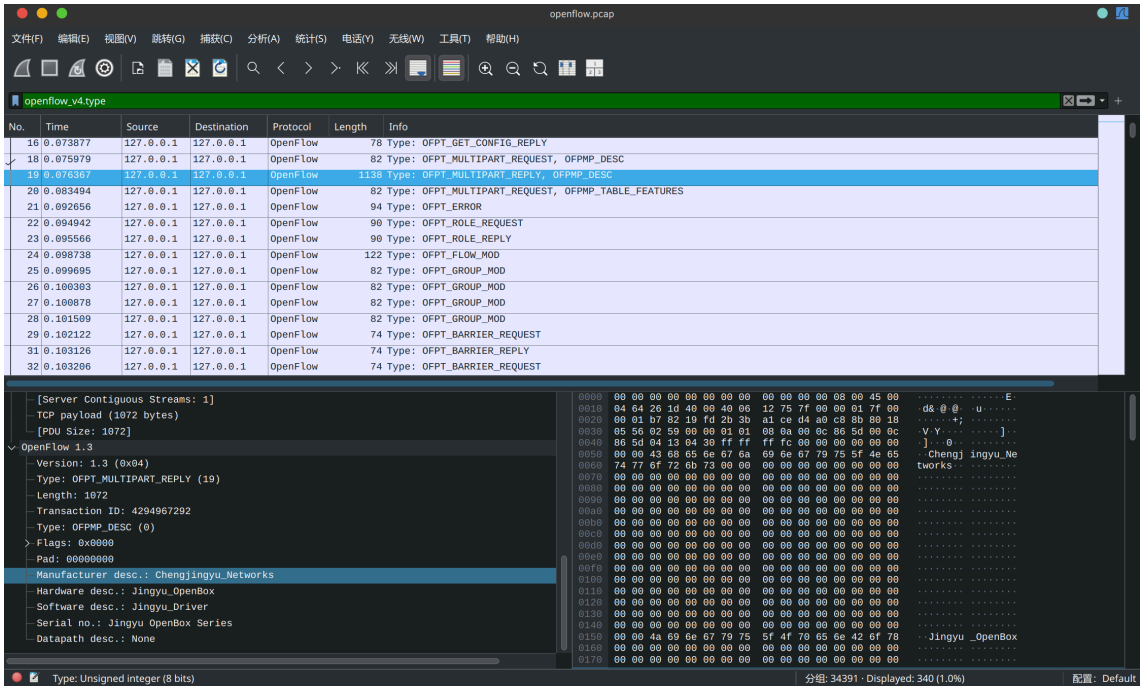

Figure 3: OFPMP_DESC 消息

可以看到图中的消息包含了制造商、硬件版本、软件版本、序列号和数据路径描述等信息（此处使用了自己的信息进行标记与区分）。

OFPMP_TABLE 类型消息。此消息用于查询交换机的流表统计信息，包括匹配计数、查找计数等。

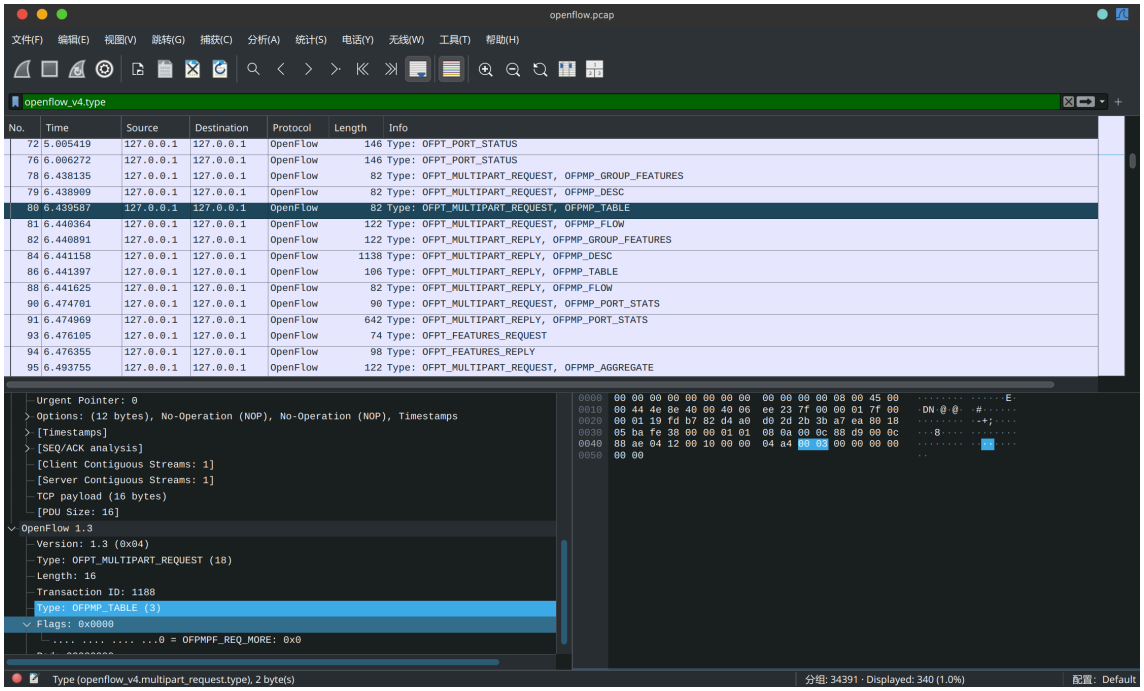

Figure 4: OFPMP_TABLE 消息

可以看到图中的消息包含了匹配计数、查找计数等信息（此处修改为了自己设置的值）。

3 实验总结

本次实验通过编写 SDN 交换机源码处理 OpenFlow 协议消息，深入理解了 OpenFlow 协议的基本消息格式和处理流程。实验涵盖了从交换机特性查询到流表统计、端口统计等多种消息类型的处理，成功实现了大部分 OpenFlow 协议的核心功能。通过实验，我不仅掌握了 OpenFlow 协议的基本工作原理，还熟悉了如何通过代码实现协议消息的解析与响应。

在实验过程中，我首先熟悉了 Openbox-S4 的基本功能，并通过编写代码实现了对 OpenFlow 协议消息的处理。实验代码中，我实现了包括 Hello 消息、Features Request 消息、Flow Stats Request 消息等在内的多种消息处理函数。每个函数都根据消息类型生成相应的回复消息，并通过 `send_openflow_message` 函数发送给控制器。通过抓包分析，我验证了消息的交互过程，确保协议编写的正确性。

实验中的难点在于如何正确处理不同类型的 OpenFlow 消息，并确保消息的格式和内容符合协议规范。通过查阅 OpenFlow 协议规范和参考相关文献，我逐步解决了这些问题，并成功实现了协议的核心功能。

实验结果表明，我编写的代码能够正确处理控制器发送的 OpenFlow 消息，并返回相应的回复消息。通过抓包分析，我验证了消息的交互过程，确保协议编写的正确性。实验的成功不仅加深了我对 OpenFlow 协议的理解，也提升了我的编程能力和网络协议分析能力。

总的来说，本次实验达到了预期的目标，成功实现了 OpenFlow 协议的核心功能。通过实验，我不仅掌握了 OpenFlow 协议的基本工作原理，还熟悉了如何通过代码实现协议消息的解析与响应。未来，我可以在此基础上进一步扩展功能，如实现流表规则的动态添加和删除，以更好地支持 SDN 网络的灵活性和可扩展性。

4 选做实验介绍

本小组了解了虚拟网络拓扑实验和 DDoS 网络攻击的 SDN 实验的基本内容，但由于时间和精力有限，最终选择只完成 DDoS 网络攻击的 SDN 实验。以下是对该实验的简要介绍。

4.1 实验原理

本实验通过搭建一个基于 SDN 的网络环境，利用 OpenBox-S4 设备启动 OVS 交换机、sFlow-RT 流量监控和 Floodlight 控制器，模拟 DDoS 攻击并实现防御。实验的核心原理是通过 SDN 控制器（Floodlight）动态下发流表规则，检测并阻断异常流量。

在实验中，主机 A 作为攻击者，使用 DDoS 攻击工具向主机 B（靶机）发送大量请求数据包，模拟 DDoS 攻击。sFlow-RT 用于实时监控网络流量，检测异常流量模式。Floodlight 控制器根据 sFlow-RT 的检测结果，下发流表规则到 OVS 交换机，丢弃匹配的异常流量，从而实现对 DDoS 攻击的防御。

实验的关键在于通过 SDN 控制器的集中控制能力，动态调整网络流表，快速响应并阻断攻击流量。这种基于 SDN 的防御机制能够有效应对 DDoS 攻击，减少对网络资源的消耗，并提高网络的整体安全性。

实验步骤参考《SDN 网络-DDOS 攻击案例操作手册.pdf》。下面仅列出部分步骤或结果，供参考。

4.1.1 发起 DDoS 攻击

1.在主机 A 上使用攻击工具或脚本发起 DDoS 攻击，向主机 B 发送大量请求数据包。 2.监控 sFlow-RT 界面，观察网络流量的变化，检测到异常流量模式。

Figure 5: 遭受 DDoS 攻击

3.记录攻击开始时间和网络状态。 4.查看流表项，发现大量异常流量匹配规则。

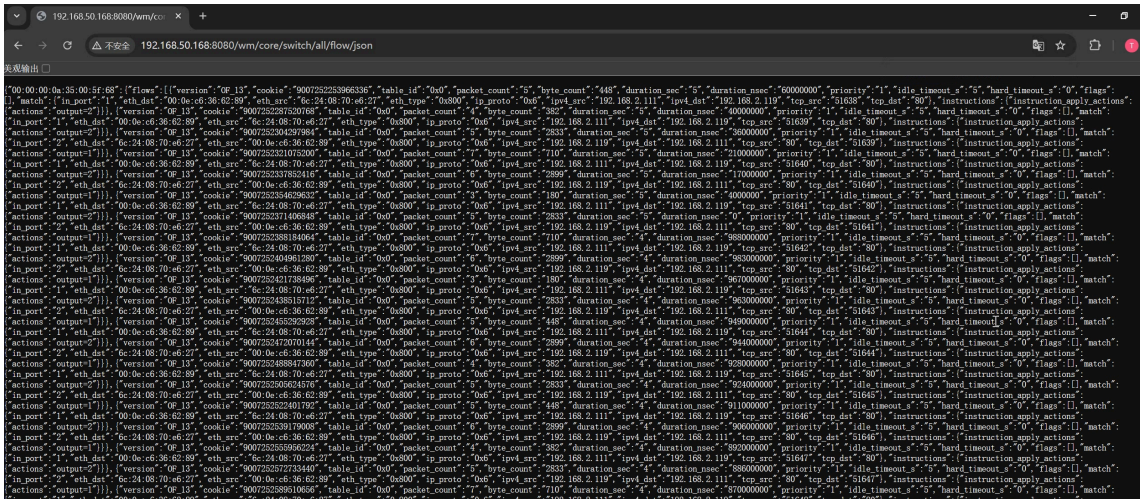

Figure 6: 流表项

4.1.2 防御 DDoS 攻击

1.使用 Floodlight 控制器下发流表规则，阻断异常流量：

```
import httplib
import json

class StaticFlowPusher(object):
    def __init__(self, server):
        self.server = server

    def get(self, data):
        ret = self.rest_call({}, 'GET')
        return json.loads(ret[2])

    def set(self, data):
        ret = self.rest_call(data, 'POST')
        return ret[0] == 200

    def remove(self, objtype, data):
        ret = self.rest_call(data, 'DELETE')
        return ret[0] == 200

    def rest_call(self, data, action):
        path = '/wm/staticflowpusher/json'
        headers = {
            'Content-type': 'application/json',
            'Accept': 'application/json',
        }
```

```
        body = json.dumps(data)
        conn = httplib.HTTPConnection(self.server, 8080)
        conn.request(action, path, body, headers)
        response = conn.getresponse()
        ret = (response.status, response.reason, response.read())
        print ret
        conn.close()
        return ret

pusher = StaticFlowPusher('127.0.0.1')

flowbe2 = {
    'switch': "00:00:00:0a:35:00:5f:68",
    "name": "flow-2",
    "cookie": "60",
    "priority": "100",
    "active": "true",
    "eth_type": "0x800",
    "in_port": "0",
    "ipv4_src": "192.168.2.111",
    "ipv4_dst": "192.168.2.119",
    "actions": "drop"
}

pusher.set(flowbe2)
```

其中关键部分在于下发流表规则，将匹配的数据包丢弃（`"actions": "drop"`）。2.检查流量图，发现异常流量被丢弃。

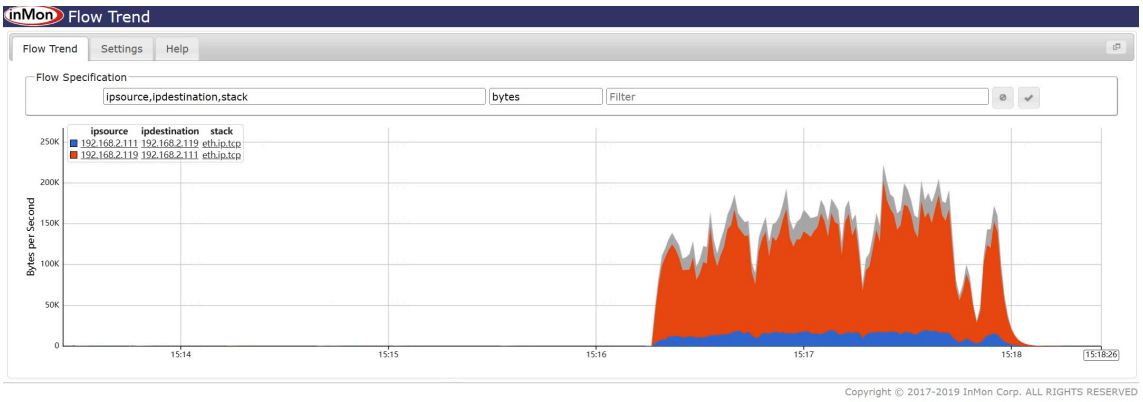

Figure 7: 丢弃异常流量

3.检查流表项，发现异常流量匹配规则消失。

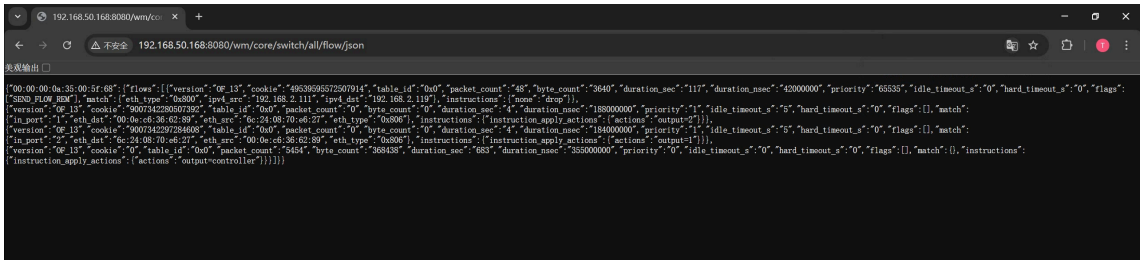

Figure 8: 流表项

4.2 选做实验总结

本次实验通过搭建基于 OpenBox-S4 设备的 SDN 网络环境，模拟 DDoS 攻击和防御过程。通过实验，我了解了 sFlow-RT 的配置和使用方法，掌握了基于 Floodlight 控制器的 DDoS 防御原理和方法。实验中，我成功模拟了 DDoS 攻击，观察了网络流量的变化，学会了使用 Floodlight 控制器下发流表规则，阻断异常流量。通过实验，我深入理解了 SDN 网络中 DDoS 攻击的特点和防御策略，提高了网络安全意识和实践能力。

参考文献

[1] OPEN NETWORKING FOUNDATION. OpenFlow Switch Specification Version 1.3.0[EB/OL]. (2014-10). https://opennetworking.org/wp-content/uploads/2014/10/openflow-spec-v1.3.0.pdf.

[2] GOLDSUNSHINE. OpenFlow 协议详解[EB/OL]. (2017-07-28). https://www.cnblogs.com/goldsunshine/p/7262484.html.

[3] 王小二. OpenFlow 协议简介[EB/OL]. (2024-12-25). https://www.jianshu.com/p/acfeae1771b3.

[4] CODEJUNERY2022. 深入理解 OpenFlow 协议[EB/OL]. (2024-12-25). https://www.jianshu.com/p/82e238eb8d14.

[5] 湖南新实网络科技有限公司. 产品上电检测手册[EB/OL]. 湖南长沙岳麓区中电软件园 6 栋 302 室: 湖南新实网络科技有限公司, 2020. http://www.xperis.com.cn/.

[6] 基于 OpenFlow 的 SDN 交换机[A]//未知. SDN 技术与应用. 未知, 2025.

[7] 湖南新实网络科技有限公司. SDN 网络-DDOS 攻击实验操作手册[EB/OL]. 湖南长沙岳麓区中电软件园 6 栋 302 室: 湖南新实网络科技有限公司, 2023. http://www.xperis.cn/.

[8] 湖南新实网络科技有限公司. SDN 网络-DDOS 攻击实验指导书[A/OL]. 湖南长沙岳麓区中电软件园 6 栋 302 室: 湖南新实网络科技有限公司, 2023. http://www.xperis.com.cn/.