一、实验目的

理解网络中网络故障出现的必然性
理解网络验证工具VeriFlow的原理
掌握VeriFlow的检测网络故障的方法
提高阅读工程代码、修改代码的能力

二、实验环境

实验在VMware运行的Arch Linux虚拟机上进行

组件	版本/配置
控制器	Ryu
虚拟化平台	Mininet
SDN网络验证工具	Veriflow

三、热身内容

观察现象

1. 启动拓扑

```
sudo ./simple.py
```

2. 启动控制程序

```
TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager
ryu.app.ofctl_rest as_switch.py
```

3. 在拓扑中ucla2 ping purdue建立连接

```
mininet> ucla2 ping purdue
```

发现ping不通。查看控制程序，发生NameResoulutionError:Failed to resolve 'localhost' ([Errno -2] No address found)
而启动控制程序时提示

```
wsgi starting up on http://0.0.0.0:8080
```

报错信息提示

```
File "/home/sdn/sdn-lab4/utils/flowmod.py", line 35, in send_flow_mod
```

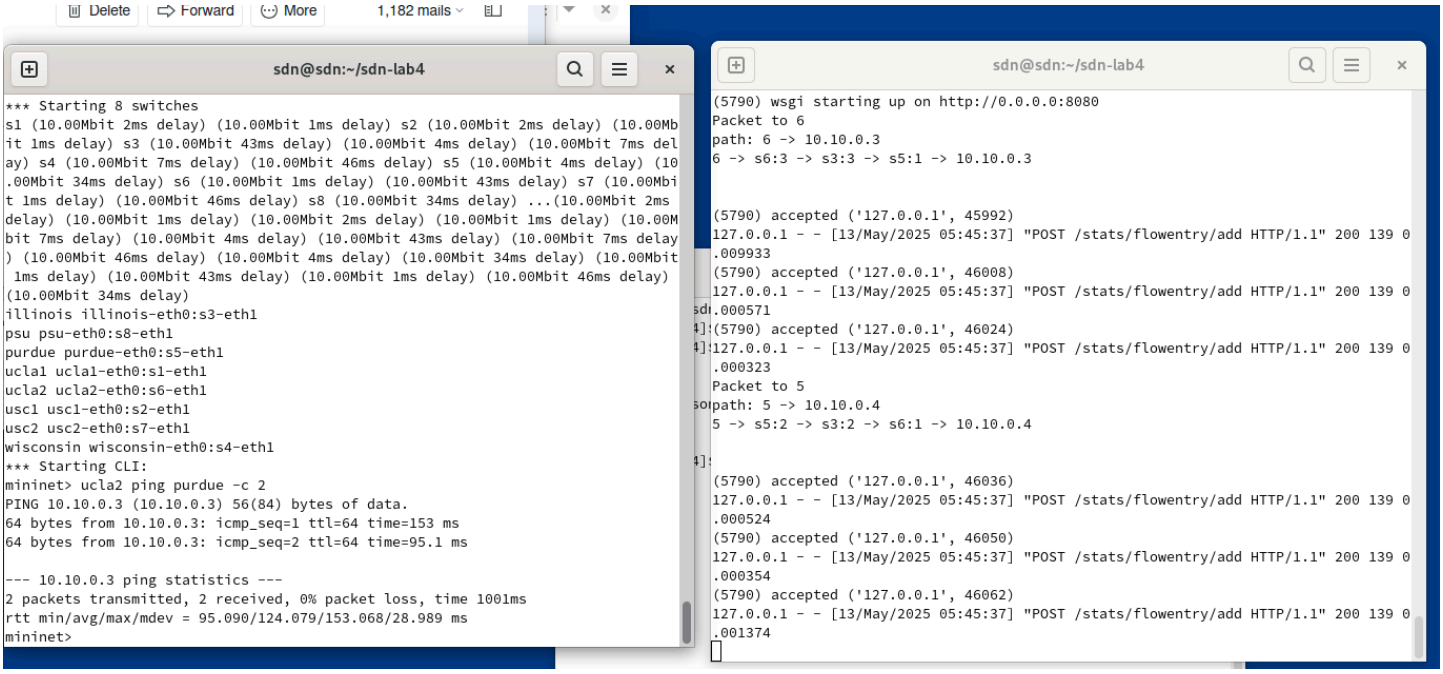
打开flowmod.py,找到错误位置。

```
url = "http://localhost:8080/stats/flowentry/add"
```

修改为

```
url = "http://0.0.0.0:8080/stats/flowentry/add"
```

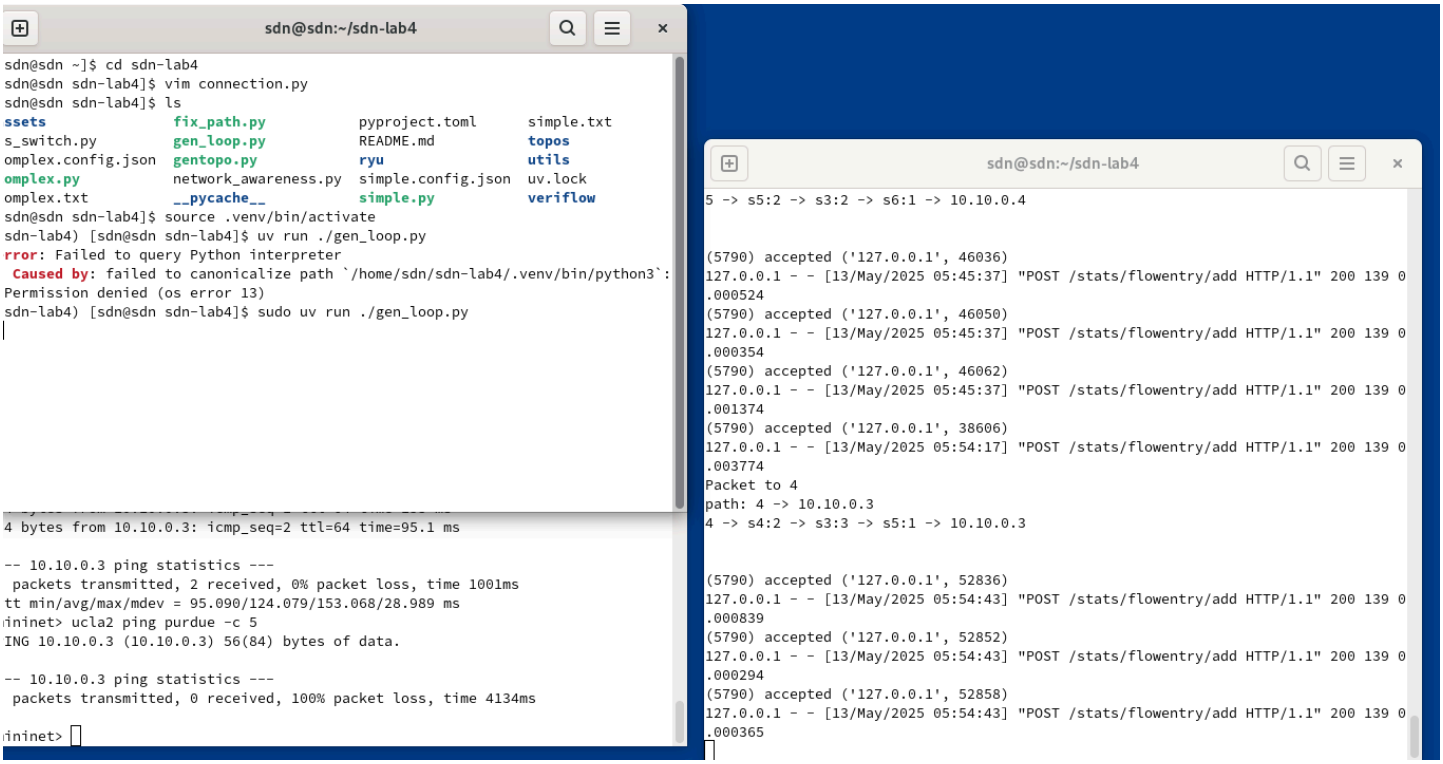
再次尝试，发现ping通，建立连接。



4. 接着下发从illinois途经wisconsin到达ucla2的路径之后，尝试ucla2 ping purdue失败

```
uv run ./gen_loop.py  #下放从illinois途经wisconsin到达ucla2的路径
```

再次尝试ucla2 ping purdue,发现失败。



5. 查看路径上某一个交换机，如illinois的流表，发现匹配某一条流表的数据包数目异常增加

```
sudo ovs-ofctl dump-flows s3
```

```
(sdn-lab4) [sdn@sdn sdn-lab4]$ sudo ovs-ofctl dump-flows s3
 cookie=0x0, duration=174.321s, table=0, n_packets=5, n_bytes=490, priority=10,i
p,nw_dst=10.10.0.0/16 actions=output:"s3-eth4"
 cookie=0x0, duration=693.768s, table=0, n_packets=2, n_bytes=196, priority=5,ip
,nw_src=10.10.0.3,nw_dst=10.10.0.4 actions=output:"s3-eth2"
 cookie=0x0, duration=148.384s, table=0, n_packets=0, n_bytes=0, priority=5,ip,n
w_src=10.10.0.4,nw_dst=10.10.0.3 actions=output:"s3-eth3"
 cookie=0x0, duration=700.312s, table=0, n_packets=28, n_bytes=1960, priority=0
actions=CONTROLLER:65509
(sdn-lab4) [sdn@sdn sdn-lab4]$
```

使用VeriFlow验证环路问题

1. 编译VeriFlow.

```
make -C veriflow -j$(nproc)
```

2. 在自定义端口开启远程控制器，运行AS控制器程序

```
TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager
ryu.app.ofctl_rest as_switch.py --ofp-tcp-listen-port 1024
```

3. 运行VeriFlow的proxy模式

将simple.txt移入到veriflow文件夹中。

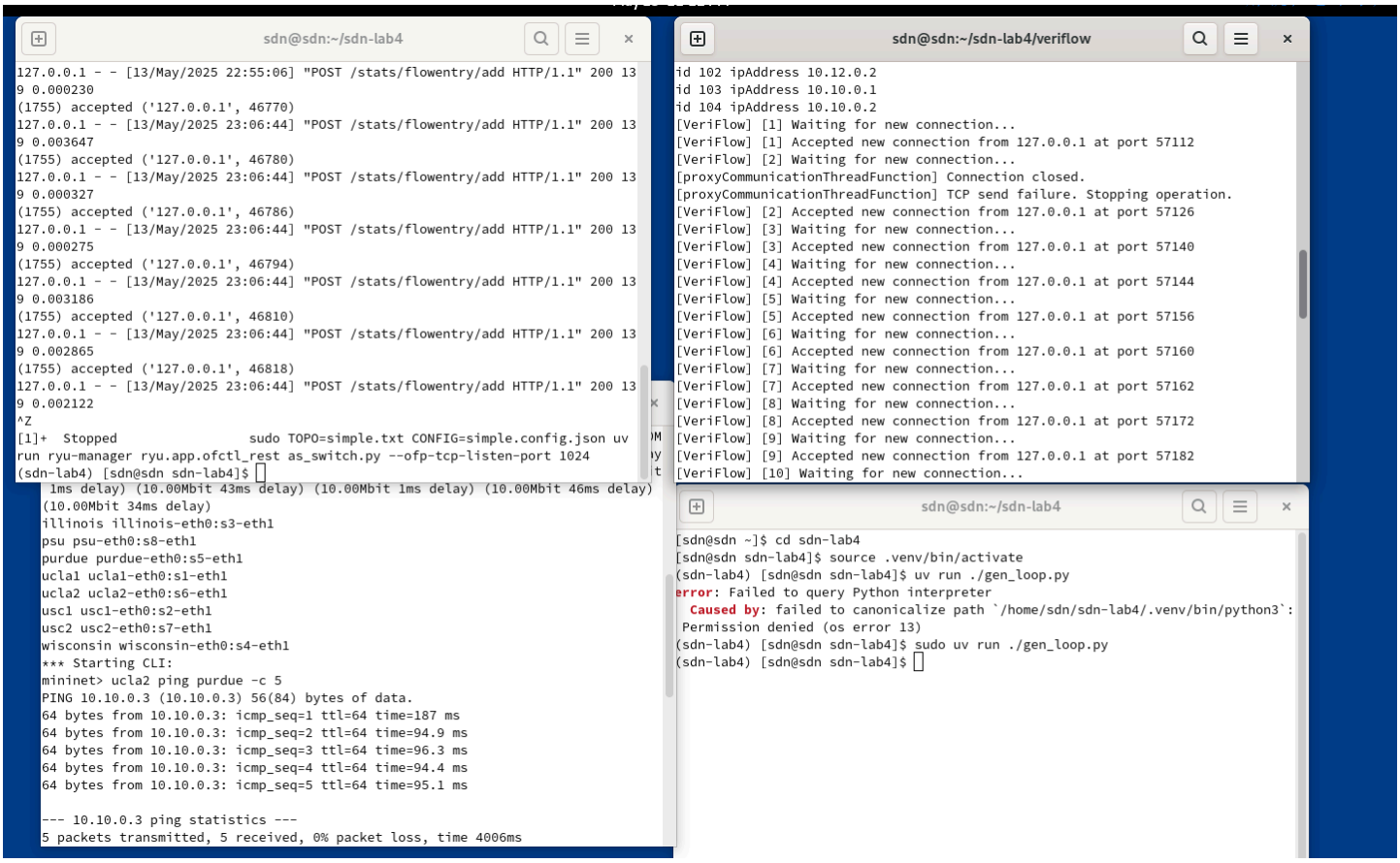
```
veriflow/VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log
```

4. 启动拓扑

```
sudo ./simple.py
```

5. 在拓扑中ucla2 ping purdue建立连接

```
mininet> ucla2 ping purdue
```



6. 下发从illinois途经wisconsin到达ucla2的路径，在log文件中观察VeriFlow检测到的环路信息。

```
uv run ./gen_loop.py
```

这里发现没有相关输出，需要完成热身内容。

1.输出每次影响EC的数量。
2.打印出环路路径的信息。
3.进一步打印出环路对应的EC的相关信息。仅展示源MAC、目的MAC和TCP/IP五元组(即源IP、目的IP、协议、源端口、目的端口)。

EC数目的打印

对VeriFlow.cpp的verifyRule函数进行修改

```
if(ecCount == 0)
{
    fprintf(stderr, "[VeriFlow::verifyRule] Error in rule: %s\n", rule.toString().c_str());
    fprintf(stderr, "[VeriFlow::verifyRule] Error: (ecCount = vFinalPacketClasses.size() = 0). Terminating proc
    exit(1);
}
else
{
    // fprintf(stdout, "\n");
    // fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
}
```

修改为

```
if(ecCount == 0)
{
    fprintf(stderr, "[VeriFlow::verifyRule] Error in rule: %s\n", rule.toString().c_str());
    fprintf(stderr, "[VeriFlow::verifyRule] Error: (ecCount = vFinalPacketClasses.size() = 0). Terminating proc
    exit(1);
}
else
{
    fprintf(stdout, "\n");
    fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
    fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
}
```

环路路径的信息的打印

VeriFlow::traverseForwardingGraph()遍历某个特定 EC 的转发图，验证是否存在环路或黑洞。函数中的变量visited记录遍历的节点，但visited是unordered_set类型，无法通过visited获得环路路径，因此我们需要加入新变量path用来记录环路，并在之前visited更新的位置同步更新，之后在检测到loop的时候遍历path来输出环路路径即可。

对VeriFlow.cpp的traverseForwardingGraph函数中检测环路的部分进行修改

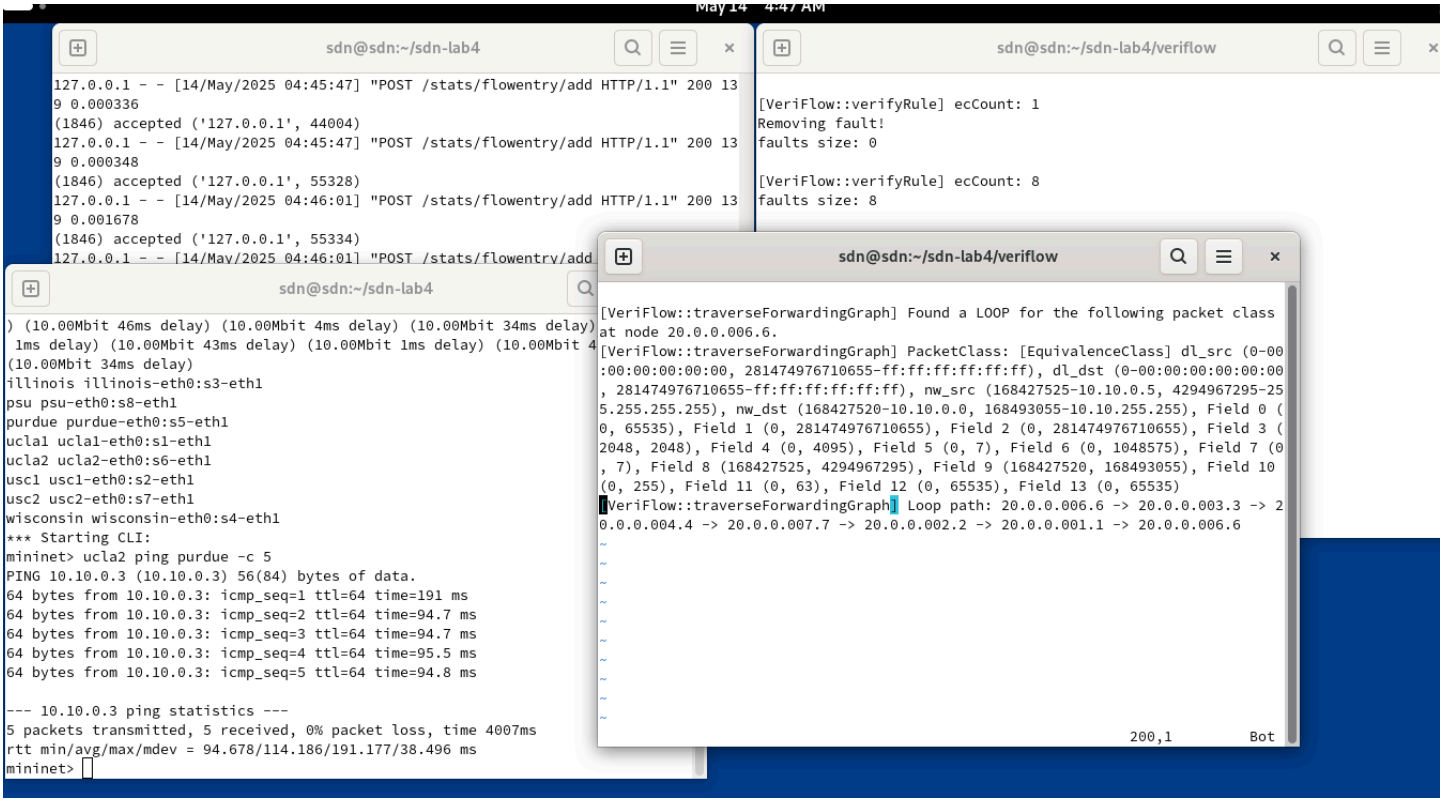
```
...
    if(visited.find(currentLocation) != visited.end())
    {
                // Found a loop.
                fprintf(fp, "\n");
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_st

                fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path: ");
                for(unsigned int i = 0; i < path.size(); i++)
                        fprintf(fp, "%s -> ", path[i].c_str());
                fprintf(fp, "%s\n", currentLocation.c_str());

                fprintf(stdout, "\n");
                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class
                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString()

                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] Loop path: ");
                for(unsigned int i = 0; i < path.size(); i++)
                        fprintf(stdout, "%s -> ", path[i].c_str());
                fprintf(stdout, "%s\n", currentLocation.c_str());

                vector<EquivalenceClass> newFaults;
                for(unsigned int i = 0; i < faults.size(); i++) {
                        EquivalenceClass& fault = faults[i];
                        if (fault.intersects(packetClass)) {
                                vector<EquivalenceClass> diff = fault.subtract(packetClass);
                                for (auto&ec : diff) {
                                        newFaults.push_back(ec);
                                }
                        }
                        else {
                                newFaults.push_back(fault);
                        }

                }
                faults = newFaults;
                faults.push_back(packetClass);

                return false;
        }

        visited.insert(currentLocation);
        path.push_back(currentLocation);
...
```

与环路对应的EC的相关信息打印

EC的基本信息打印在VeriFlow::traverseForwardingGraph()中通过调用EC类的toString()函数实现。
现在将EC类的toString()函数注释掉，重新写一个toString函数即可。

```
string EquivalenceClass::toString() const
{
        char buffer[1024];

        sprintf(buffer, "[EquivalenceClass] dl_src (%s, %s), dl_dst (%s, %s)",
                        ::getMacValueAsString(this->lowerBound[DL_SRC]).c_str(),
                        ::getMacValueAsString(this->upperBound[DL_SRC]).c_str(),
                        ::getMacValueAsString(this->lowerBound[DL_DST]).c_str(),
                        ::getMacValueAsString(this->upperBound[DL_DST]).c_str());
        string retVal = buffer;
        retVal += ", ";

        sprintf(buffer, "[EquivalenceClass] nw_src (%s, %s), nw_dst (%s, %s)",
                        ::getIpValueAsString(this->lowerBound[NW_SRC]).c_str(),
                        ::getIpValueAsString(this->upperBound[NW_SRC]).c_str(),
                        ::getIpValueAsString(this->lowerBound[NW_DST]).c_str(),
                        ::getIpValueAsString(this->upperBound[NW_DST]).c_str());
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "nw_proto(%lu, %lu)", this->lowerBound[NW_PROTO], this->upperBound[NW_PROTO]);
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "tp_src(%lu, %lu)", this->lowerBound[TP_SRC], this->upperBound[TP_SRC]);
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "tp_dst(%lu, %lu)", this->lowerBound[TP_DST], this->upperBound[TP_DST]);
        retVal += buffer;
        retVal += ", ";

        return retVal;
}
```

热身实验结果展示



四、必做任务描述及方案设计与结果

解决了刚才的转发环路问题后，你想尝试跨AS的数据包转发是否存在一些问题，于是决定测试ucla1和illinois的连通性。然而，你却发现，VeriFlow验证工具出现了一些问题。

1. 在自定义端口开启远程控制器，运行AS控制器程序

```
TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager
ryu.app.ofctl_rest as_switch.py --ofp-tcp-listen-port 1024
```

2. 运行VeriFlow的proxy模式：

```
veriflow/VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log
```

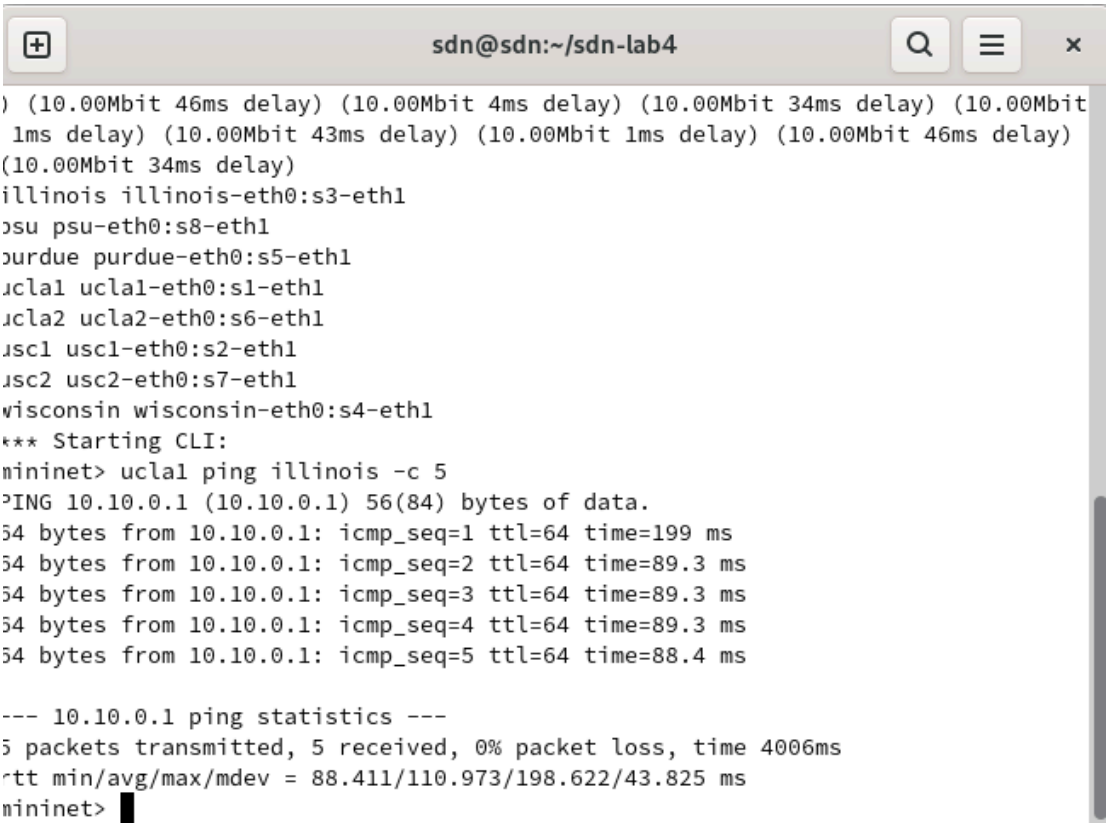
3. 启动拓扑

```
sudo ./simple.py
```

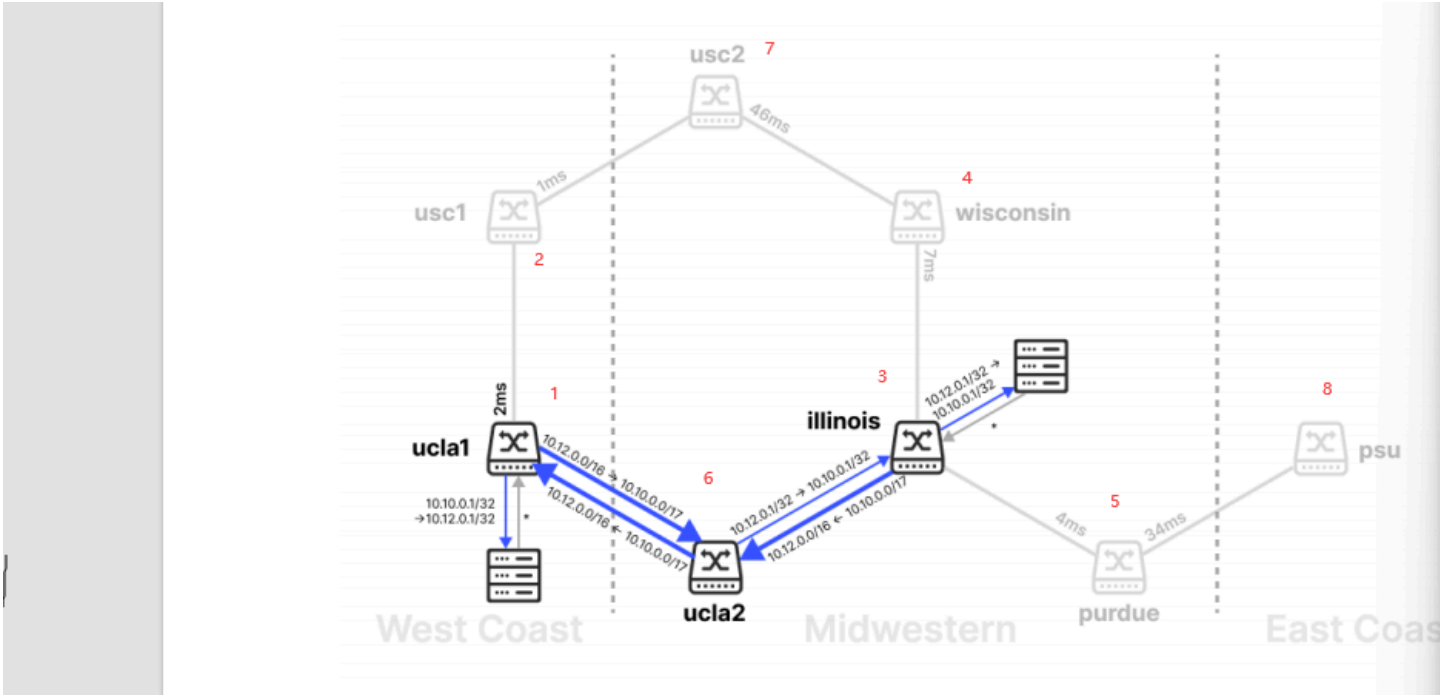
4. 建立转发路径

```
mininet> ucla1 ping illinois
```

此时，两者可以正常ping通，但veriflow提示出现了黑洞。



连通性测试后的流表示意图如下：



分析黑洞的原因

ucla1交换机 ---(10.10.0.1/32 -> 10.12.0.1/32) ---> ucla1连接的主机

ucla1交换机 --- (10.12.0.0/16 -> 10.10.0.0/17) ---> ucla2 交换机

ucla2交换机 --- (10.12.0.1/32 -> 10.10.0.1/32) ---> illinois 交换机

illinois交换机 --- (10.12.0.1/32 -> 10.10.0.1/32) ---> illinois连接的主机

illinois交换机 --- (10.12.0.0/16 -> 10.10.0.0/17) ---> ucla2 交换机

ucla2交换机 --- (10.12.0.0/16 -> 10.10.0.0/17) ---> ucla1 交换机

主机名	交换机	dpid	IP地址	所属网段
ucla1	s1	1	10.12.0.1	10.12.0.0/16
usc1	s2	2	10.12.0.2	10.12.0.0/16
illinois	s3	3	10.10.0.1	10.10.0.0/17
wisconsin	s4	4	10.10.0.2	10.10.0.0/17
purdue	s5	5	10.10.0.3	10.10.0.0/17
ucla2	s6	6	10.10.0.4	10.10.0.0/17
usc2	s7	7	10.10.0.5	10.10.0.0/17
psu	s8	8	10.10.128.1	10.10.128.0/17

找出所有EC

EC1

(10.10.0.1/32 → 10.12.0.1/32)

EC2

(10.12.0.0/16 → 10.10.0.0/17)

EC3

(10.12.0.1/32 → 10.10.0.1/32)

EC4

(10.10.0.0/17 → 10.12.0.0/16)

对于可能的转发路径上的每个交换机，对每个EC进行分析

ucla1交换机

(10.10.0.1/32 → 10.12.0.1/32) → ucla1主机
(10.12.0.0/16 → 10.10.0.0/17) → ucla2交换机

显然EC1 和 EC2在ucla1 交换机上都有流表项匹配
EC3 匹配第二条
EC4 在ucla1无匹配 黑洞(10.10.0.0/17 → 10.12.0.0/16)

ucla2交换机

(10.12.0.1/32 → 10.10.0.1/32) → illinois交换机
(10.10.0.0/17 → 10.12.0.0/16) → ucla1交换机

EC3,4有匹配，EC1有匹配
EC2无匹配(10.12.0.0/16 → 10.10.0.0/17)

illinois交换机

(10.12.0.1/32 → 10.10.0.1/32) → illinois主机
(10.10.0.0/17 → 10.12.0.0/16) → ucla2交换机

EC3,4有匹配，EC1有匹配
EC2无匹配(10.12.0.0/16 → 10.10.0.0/17)

5.运行uv run ./fix_path.py

```
#!/usr/bin/env python3
from utils.flowmod import send_flow_mod

send_flow_mod(6, None, None, None, '10.10.0.0/16', None, 3)
send_flow_mod(3, None, None, None, '10.10.0.0/16', None, 1)
send_flow_mod(1, None, None, None, '10.12.0.0/16', None, 1)
```

因此给ucla2和illinois交换机下放匹配dst为10.10.0.0/16
给ucla1交换机下放匹配dst为10.12.0.0/16
即可所有EC无黑洞

新增的2个EC dst为10.10.0.0/16与dst为10.12.0.0/16
dst为10.12.0.0/16在ucla2与illinois都有匹配流表项，

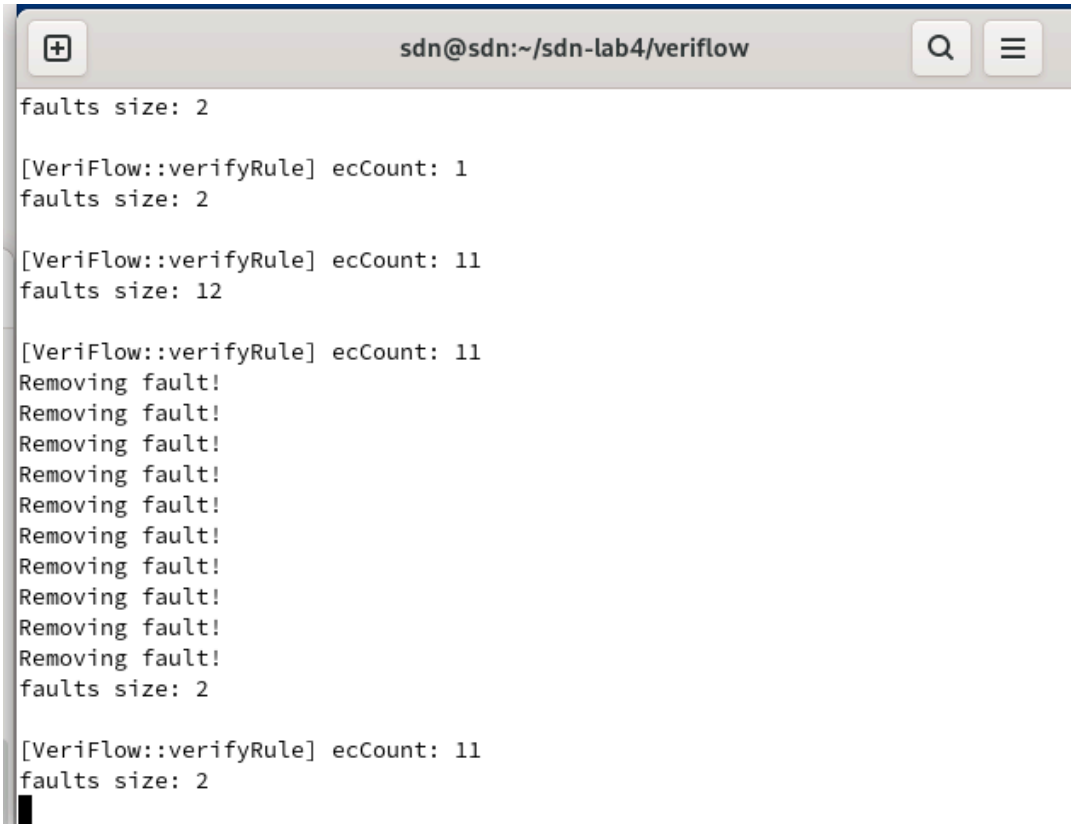
而dst为10.10.0.0/16
其中dst为10.10.0.0/16的EC在ucla1交换机上并无匹配流表项。

其中ucla2的dst为10.10.0.0/16的转发到端口3 s3 ill

illinois的dst为10.10.0.0/16的转发到端口1 ill

ucla1的dst为10.12.0.0/16的转发到端口1 ucla1
所以新EC也不会有黑洞。

6.发现veriflow依然有网络错误



这是因为，在veriflow代码中，对新规则影响错误数量计算有错误。仅仅考虑了新规则的等价类完全包含错误的等价类的情况；此时，如果一个新规则的等价类覆盖了多个之前的错误，那么由于原来错误均不完全被该等价类所包含，旧的错误会依然存在；导致错误数量的计算可能有误。

必做任务方案设计

通过fault和packetClass是否有交集，判断是否影响到旧fault。若有交集，那fault要将除交集以外的部分删除，若删除后是空的，则将fault去除。

为此，我们要在EquivalenceClass类中实现两个方法：一是判断交集，二是求差。

求交集

在EquivalenceClass.h增加声明

```
bool intersects(const EquivalenceClass& other) const;
```

在EquivalenceClass.cpp中实现

```
bool EquivalenceClass::intersects(const EquivalenceClass& other) const {
        for (int i=0; i< ALL_FIELD_INDEX_END_MARKER; i++){
                if (this->upperBound[i] < other.lowerBound[i] || this->lowerBound[i] > other.upperBound[i]){
                        return false;
                }
        }
        return true;
}
```

代码含义：若有一个属性无交集，则fault与EC无交集。例如src 0-1 与 src 2-3的fault和EC，无论其他属性是否有交集，都不会相交。

求差

在EquivalenceClass.h增加声明

```
        vector<EquivalenceClass> subtract(const EquivalenceClass& other) const;
```

在EquivalenceClass.cpp中实现

```
vector<EquivalenceClass> EquivalenceClass::subtract(const EquivalenceClass& other) const {
        vector<EquivalenceClass> result;

        // 如果不相交，直接返回当前等价类
        if (!this->intersects(other)) {
                result.push_back(*this);
                return result;
        }

        // 创建一个临时等价类用于存储中间结果
        EquivalenceClass temp = *this;

        // 对每个字段进行处理
        for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
                // 如果当前字段有重叠
                if (temp.lowerBound[i] <= other.upperBound[i] && temp.upperBound[i] >= other.lowerBound[i]) {
                        // 处理下界部分
                        if (temp.lowerBound[i] < other.lowerBound[i]) {
                                EquivalenceClass left = temp;
                                left.upperBound[i] = other.lowerBound[i] - 1;
                                result.push_back(left);
                        }

                        // 处理上界部分
                        if (temp.upperBound[i] > other.upperBound[i]) {
                                EquivalenceClass right = temp;
                                right.lowerBound[i] = other.upperBound[i] + 1;
                                result.push_back(right);
                        }

                        // 更新临时等价类的边界为重叠部分
                        temp.lowerBound[i] = max(temp.lowerBound[i], other.lowerBound[i]);
                        temp.upperBound[i] = min(temp.upperBound[i], other.upperBound[i]);
                }
        }

        return result;
}
```

代码思路：首先调用 this->intersects(other) 判断两个等价类是否有交集。如果没有交集，直接返回当前等价类本身（即 *this），因为没有需要“减去”的部分。
创建一个临时等价类 temp，初始为当前等价类（*this），用于后续操作。
遍历每个字段，对每一个字段（如 IP、端口等，字段数量由 ALL_FIELD_INDEX_END_MARKER 决定）逐一处理。
对于每个字段，如果当前字段与 other 的对应字段有重叠（即区间有交集）：
如果当前字段的下界小于 other 的下界，说明有一部分在 other 区间左侧，这部分需要保留。
创建一个新的等价类 left，其上界设为 other.lowerBound[i] - 1，表示左侧不重叠的部分，加入结果集。
如果当前字段的上界大于 other 的上界，说明有一部分在 other 区间右侧，这部分也需要保留。
创建一个新的等价类 right，其下界设为 other.upperBound[i] + 1，表示右侧不重叠的部分，加入结果集。
最后，将 temp 的当前字段区间收缩为与 other 的重叠部分，继续处理下一个字段。
最终返回所有不与 other 重叠的等价类集合。

修改Veriflow.cpp有关部分

替换掉原来的错误代码。

原有代码（只考虑完全包含）：

```
for (unsigned int i = 0; i < faults.size(); i++) {
    if (packetClass.subsumes(faults[i])) {
        faults.erase(faults.begin() + i);
        i--;
    }
}
faults.push_back(packetClass);
```

替换为如下代码：

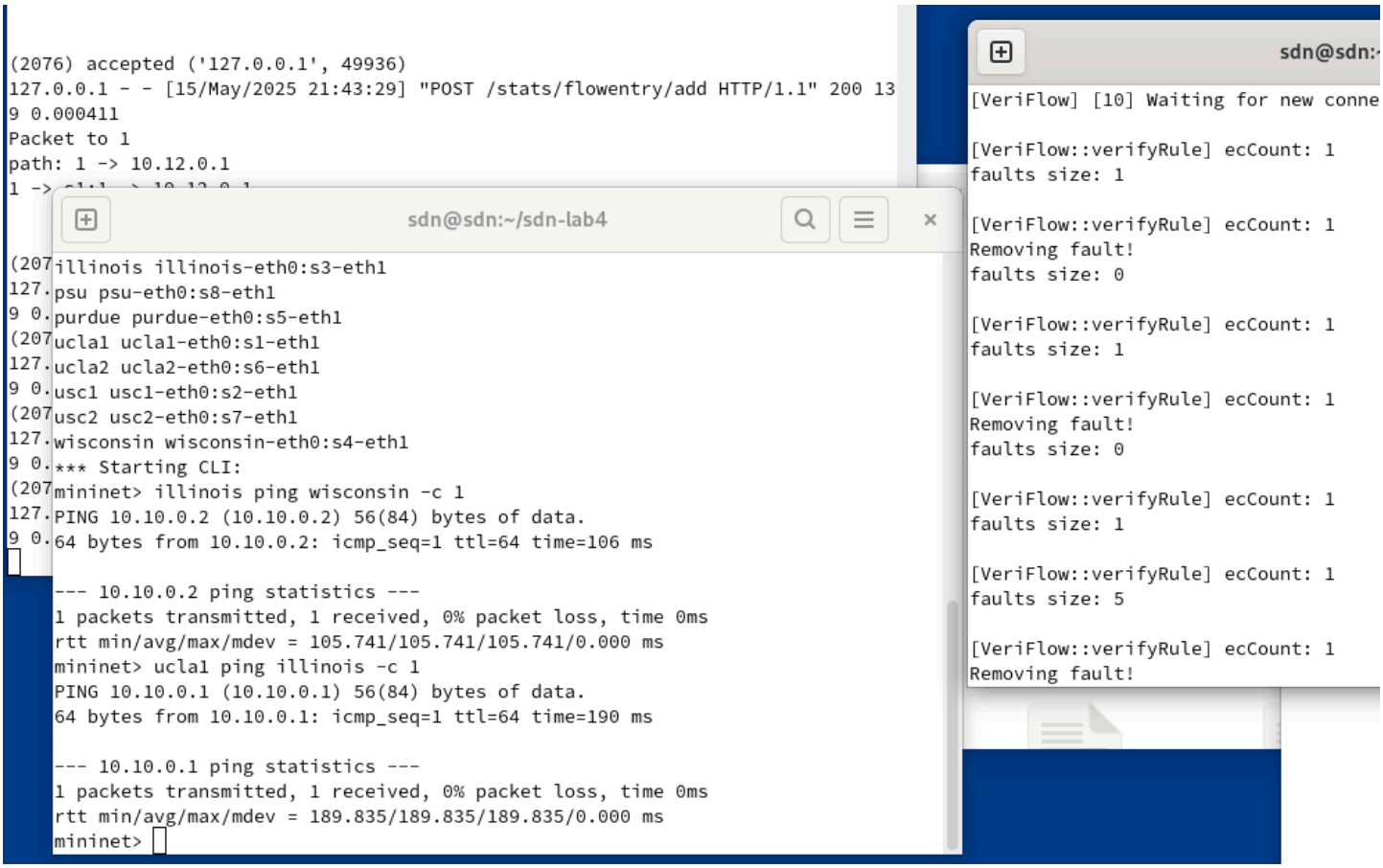
```
vector<EquivalenceClass> newFaults;

for (unsigned int i = 0; i < faults.size(); i++) {
    EquivalenceClass& fault = faults[i];
    if (fault.intersects(packetClass)) {
        // 1. 计算 fault - packetClass 的差集，可能得到多个新的EC
        vector<EquivalenceClass> diff = fault.subtract(packetClass);
        // 2. 把差集部分加入新的faults
        for (auto& ec : diff) {
            newFaults.push_back(ec);
        }
        // 3. 原fault被完全覆盖则不保留，部分覆盖则只保留未被覆盖部分
    } else {
        // 没有交集，原fault保留
        newFaults.push_back(fault);
    }
}
// 用新faults替换原faults
faults = newFaults;

// 最后把新检测到的错误EC加入faults
faults.push_back(packetClass);
```

这是检测到错误EC部分的替换，如果没检测到，而是其他的添加修改之类的更新，则不需向faults中push_back，删掉最后一行即可。

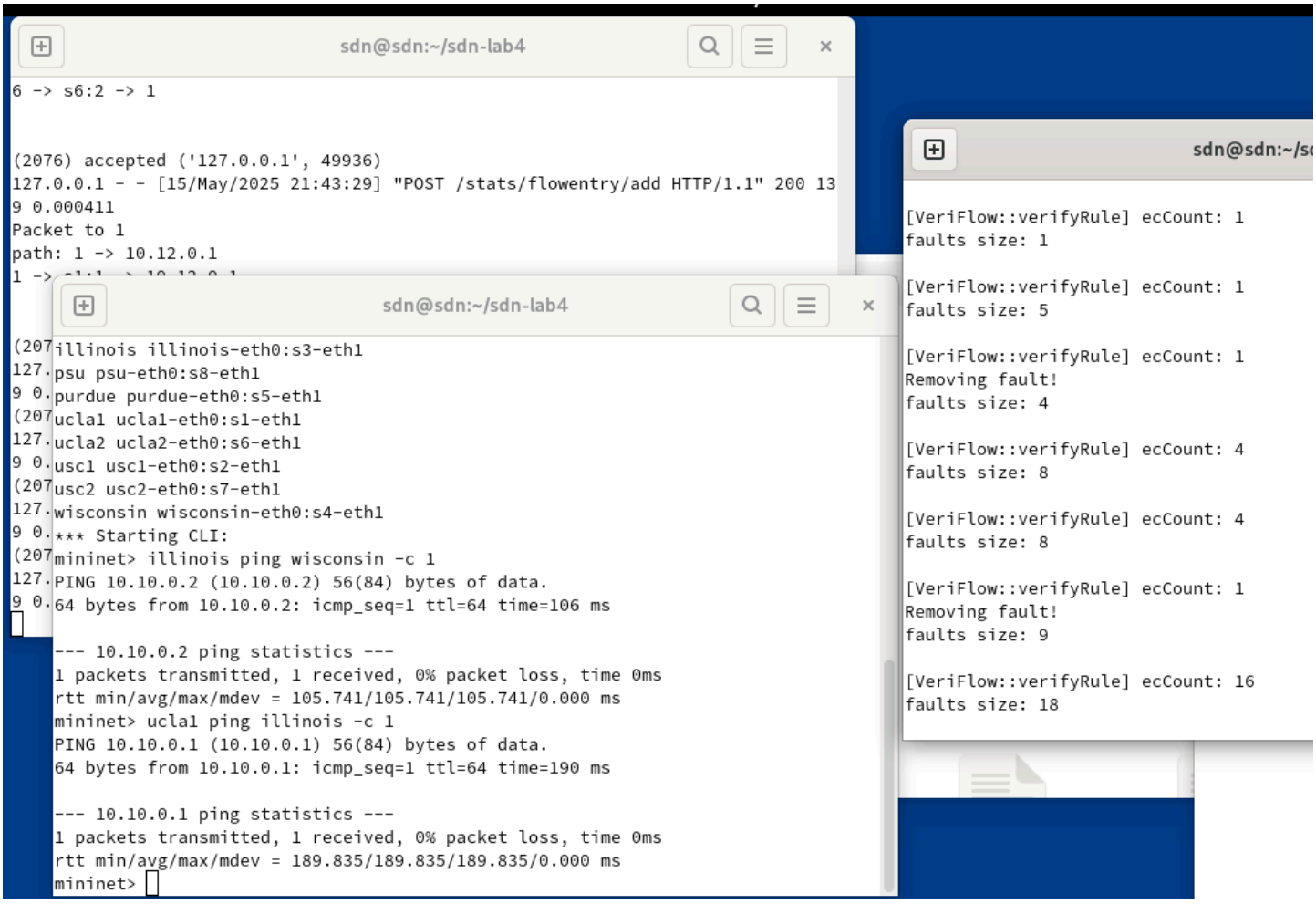
代码思路就是上述提到的。

必做任务结果

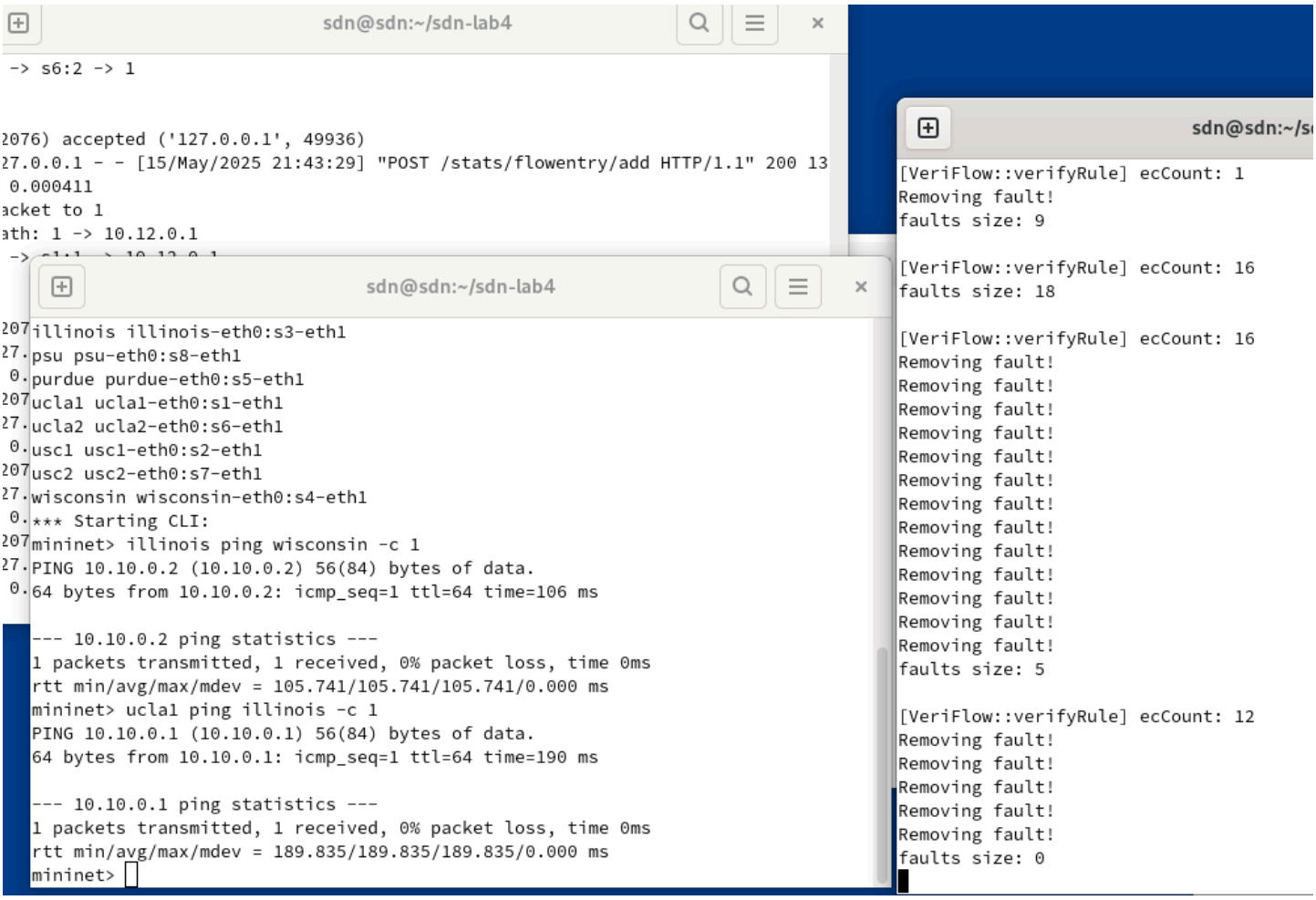
ucla1 ping illinois 之前（运行 illinois ping wisconsin ）：



ucla1 ping illinois 之后，修复之前：



修复之后：



五、选做任务

任务描述

阅读 as_switch.py 有关数据包转发的代码，解释产生黑洞的原因。请说明，产生的黑洞问题会不会影响网络的实际使用？修改 as_switch.py ，使得同样的ping操作不会再产生黑洞。

产生黑洞的原因及解决方案与结果展示

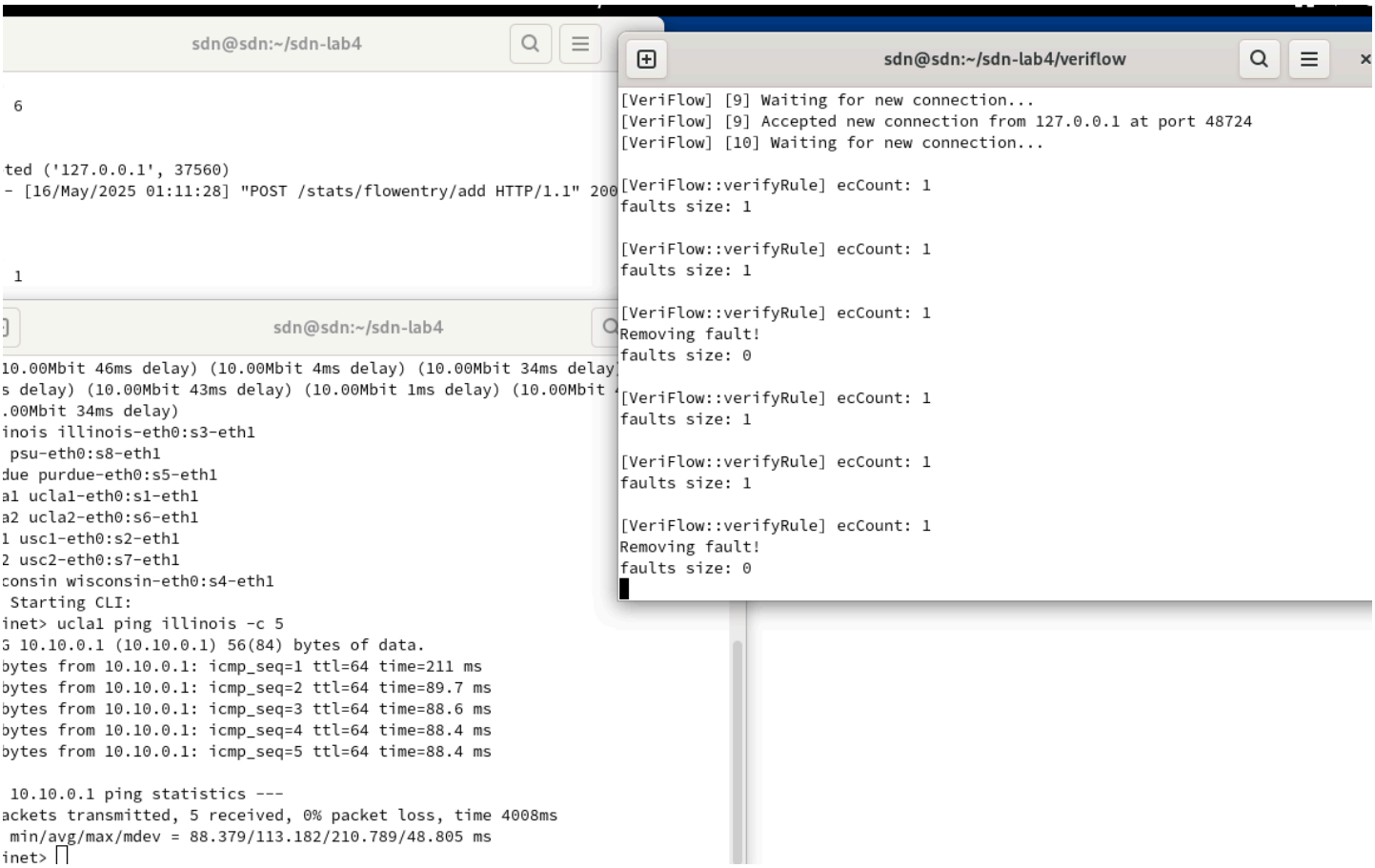
阅读as_switch.py，
发现在handle_ipv4函数中，跨as转发时，调用add_path使用了srcnet和dstnet作为匹配条件

```
out_port = add_path(route, None, None, srcnet, dstnet)
```

而as内转发则是精确匹配。

因此我们将两处均修改为精确匹配就不会产生黑洞了。

```
out_port = add_path(route, None, None, src_ip, dst_ip)
```



发现faults.size()为0，说明无黑洞产生了！

产生的黑洞问题是否会影响网络的实际使用？

不会。因为宽泛流表项匹配仅仅发生在网关交换机上，而as内的交换机到达网关交换机仍可精确匹配。
这样做可以大大减少流表项数目，提高匹配效率。

六、实验总结

在这次lab中，我验证了 SDN 网络中流表规则配置错误（如环路、黑洞）的 必然性，深刻理解动态网络环境下故障检测的必要性。通过实际案例（跨 AS 转发黑洞）发现，流表匹配范围设计与 拓扑模型一致性 对网络行为有决定性影响。
通过lab，我掌握了 VeriFlow 等价类划分（EC） 和 路径验证 的核心机制，理解其通过代理模式实现流表预验证的原理。掌握了VeriFlow的检测网络故障的方法，并提高了阅读工程代码、修改代码的能力。

附录(所有修改的代码)

In as_switch.py

```
...

    def handle_ipv4(self, msg, dpid, in_port, src_mac, dst_mac, src_ip, dst_ip, pkt_type):

        ....


        else:
            # Handle inter-AS packet switching.
            if dpid in gateways:
                # first, if self is a gateway, send to the peer
                peer = self.routing_cfg["peers"][str(dpid)][dstnet]
                route = [dpid, peer]
                # To make matching more efficient, use source net and dst net as matching condition.
                # This might match some IPs which does not exist in the network.
```

修改：

```
                out_port = add_path(route, None, None, src_ip, dst_ip)

            else:
                # otherwise, send to the closest gateway
                min_delay = math.inf
                min_gw = None

                ....

                # To make matching more efficient, use source net and dst net as matching condition.
                # This might match some IPs which does not exist in the network.
```

修改：

```
                out_port = add_path(dpid_path, None, None, src_ip, dst_ip)

...
```

In EqivalenceClass.h

```
...

class EquivalenceClass
{
public:

    ...

加入：    bool intersects(const EquivalenceClass& other) const;
加入：   vector<EquivalenceClass> subtract(const EquivalenceClass& other) const;

        ...

};
```

EqivalenceClass.cpp

In EqivalenceClass.cpp

...

加入

```
bool EquivalenceClass::intersects(const EquivalenceClass& other) const {
        for (int i=0; i< ALL_FIELD_INDEX_END_MARKER; i++){
                if (this->upperBound[i] < other.lowerBound[i] || this->lowerBound[i] > other.upperBound[i]){
                        return false;
                }
        }
        return true;
}

vector<EquivalenceClass> EquivalenceClass::subtract(const EquivalenceClass& other) const {
        vector<EquivalenceClass> result;

        // 如果不相交，直接返回当前等价类
        if (!this->intersects(other)) {
                result.push_back(*this);
                return result;
        }

        // 创建一个临时等价类用于存储中间结果
        EquivalenceClass temp = *this;

        // 对每个字段进行处理
        for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
                // 如果当前字段有重叠
                if (temp.lowerBound[i] <= other.upperBound[i] && temp.upperBound[i] >= other.lowerBound[i]) {
                        // 处理下界部分
                        if (temp.lowerBound[i] < other.lowerBound[i]) {
                                EquivalenceClass left = temp;
                                left.upperBound[i] = other.lowerBound[i] - 1;
                                result.push_back(left);
                        }

                        // 处理上界部分
                        if (temp.upperBound[i] > other.upperBound[i]) {
                                EquivalenceClass right = temp;
                                right.lowerBound[i] = other.upperBound[i] + 1;
                                result.push_back(right);
                        }

                        // 更新临时等价类的边界为重叠部分
                        temp.lowerBound[i] = max(temp.lowerBound[i], other.lowerBound[i]);
                        temp.upperBound[i] = min(temp.upperBound[i], other.upperBound[i]);
                }
        }

        return result;
}
```

```
string EquivalenceClass::toString() const
{
        char buffer[1024];

        sprintf(buffer, "[EquivalenceClass] dl_src (%s, %s), dl_dst (%s, %s)",
                        ::getMacValueAsString(this->lowerBound[DL_SRC]).c_str(),
                        ::getMacValueAsString(this->upperBound[DL_SRC]).c_str(),
                        ::getMacValueAsString(this->lowerBound[DL_DST]).c_str(),
                        ::getMacValueAsString(this->upperBound[DL_DST]).c_str());
        string retVal = buffer;
        retVal += ", ";

        sprintf(buffer, "[EquivalenceClass] nw_src (%s, %s), nw_dst (%s, %s)",
                        ::getIpValueAsString(this->lowerBound[NW_SRC]).c_str(),
                        ::getIpValueAsString(this->upperBound[NW_SRC]).c_str(),
                        ::getIpValueAsString(this->lowerBound[NW_DST]).c_str(),
                        ::getIpValueAsString(this->upperBound[NW_DST]).c_str());
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "nw_proto(%lu, %lu)", this->lowerBound[NW_PROTO], this->upperBound[NW_PROTO]);
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "tp_src(%lu, %lu)", this->lowerBound[TP_SRC], this->upperBound[TP_SRC]);
        retVal += buffer;
        retVal += ", ";

        sprintf(buffer, "tp_dst(%lu, %lu)", this->lowerBound[TP_DST], this->upperBound[TP_DST]);
        retVal += buffer;
        retVal += ", ";

        return retVal;
}
```

In VeriFlow.cpp

```
bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime, double& packetClassSearchTime, doul
{
        fprintf(fp, "[VeriFlow::verifyRule] verifying this rule: %s\n", rule.toString().c_str());


        updateTime = packetClassSearchTime = graphBuildTime = queryTime = 0;
        ecCount = 0;

        struct timeval start, end;
        double usecTime, seconds, useconds;

        gettimeofday(&start, NULL);
        // May add code in a future version to maintain a cache of forwarding graphs. This cache needs to be upd
        gettimeofday(&end, NULL);

        seconds  = end.tv_sec  - start.tv_sec;
        useconds = end.tv_usec - start.tv_usec;
        usecTime = (seconds * 1000000) + useconds;
        updateTime = usecTime;

        gettimeofday(&start, NULL);
        vector< EquivalenceClass > vFinalPacketClasses;
        vector< vector< Trie* > > vFinalTries;

        bool res = this->getAffectedEquivalenceClasses(rule, command, vFinalPacketClasses, vFinalTries);
        if(res == false)
        {
                return false;
        }
        gettimeofday(&end, NULL);

        seconds  = end.tv_sec  - start.tv_sec;
        useconds = end.tv_usec - start.tv_usec;
        usecTime = (seconds * 1000000) + useconds;
        packetClassSearchTime = usecTime;

        ecCount = vFinalPacketClasses.size();
        if(ecCount == 0)
        {
                fprintf(stderr, "[VeriFlow::verifyRule] Error in rule: %s\n", rule.toString().c_str());
                fprintf(stderr, "[VeriFlow::verifyRule] Error: (ecCount = vFinalPacketClasses.size() = 0). Term
                exit(1);
        }
        else
        {
                fprintf(stdout, "\n");
                fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
                fprintf(fp,"[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
        }

        // fprintf(stdout, "[VeriFlow::verifyRule] Generating forwarding graphs...\n");
        gettimeofday(&start, NULL);
```

```
vector< ForwardingGraph* > vGraph;
for(unsigned int i = 0; i < vFinalPacketClasses.size(); i++)
{
        EquivalenceClass packetClass = vFinalPacketClasses[i];
        // fprintf(stdout, "[VeriFlow::verifyRule] [%u] ecCount: %lu, %s\n", i, ecCount, packetClass.to
        ForwardingGraph* graph = Trie::getForwardingGraph(TP_DST, vFinalTries[i], packetClass, fp);
        vGraph.push_back(graph);
}
gettimeofday(&end, NULL);
// fprintf(stdout, "[VeriFlow::verifyRule] Generated forwarding graphs.\n");

seconds  = end.tv_sec  - start.tv_sec;
useconds = end.tv_usec - start.tv_usec;
usecTime = (seconds * 1000000) + useconds;
graphBuildTime = usecTime;

// fprintf(stdout, "[VeriFlow::verifyRule] Running query...\n");
gettimeofday(&start, NULL);
// Add query code here
size_t currentFailures = 0;
for(unsigned int i = 0; i < vGraph.size(); i++)
{
        unordered_set< string > visited;
        vector<string> path;
        string lastHop = network.getNextHopIpAddress(rule.location,rule.in_port);
        // fprintf(fp, "start traversing at: %s\n", rule.location.c_str());
        if(!this->traverseForwardingGraph(vFinalPacketClasses[i], vGraph[i], rule.location, lastHop, vi
                ++currentFailures;
        }
}

fprintf(stderr, "faults size: %li\n", faults.size());
if (previousFailures > 0 && faults.size()==0) {
        fprintf(fp, "[Veriflow::verifyRule] Network Fixed!\n");
} else if (previousFailures == 0 && faults.size() > 0) {
        fprintf(fp, "[Veriflow::verifyRule] Network Broken!\n");
}
fflush(fp);
previousFailures = faults.size();
// fprintf(stdout, "[VeriFlow::verifyRule] Query complete.\n");

if(command == OFPFC_ADD)
{
        // Do nothing.
}
else if(command == OFPFC_DELETE_STRICT)
{
        Rule dummyRule = rule;
        dummyRule.type = DUMMY;

        this->removeRule(dummyRule);
}
gettimeofday(&end, NULL);

seconds  = end.tv_sec  - start.tv_sec;
```

```
        useconds = end.tv_usec - start.tv_usec;
        usecTime = (seconds * 1000000) + useconds;
        queryTime = usecTime;

        for(unsigned int i = 0; i < vGraph.size(); i++)
        {
                delete vGraph[i];
        }

        return true;
}

bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const strin
{

        // fprintf(fp, "traversing at node: %s\n", currentLocation.c_str());
        if(graph == NULL)
        {
                /* fprintf(fp, "\n");
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] (graph == NULL) for the following packet class
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_s

                return true;
        }

        if(currentLocation.compare("") == 0)
        {
                return true;
        }

        if(visited.find(currentLocation) != visited.end())
        {
                // Found a loop.
                fprintf(fp, "\n");
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_s

                fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path: ");
                for(unsigned int i = 0; i < path.size(); i++)
                        fprintf(fp, "%s -> ", path[i].c_str());
                fprintf(fp, "%s\n", currentLocation.c_str());

                fprintf(stdout, "\n");
                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet clas
                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString()

                fprintf(stdout, "[VeriFlow::traverseForwardingGraph] Loop path: ");
                for(unsigned int i = 0; i < path.size(); i++)
                        fprintf(stdout, "%s -> ", path[i].c_str());
                fprintf(stdout, "%s\n", currentLocation.c_str());

                vector<EquivalenceClass> newFaults;
                for(unsigned int i = 0; i < faults.size(); i++) {
                        EquivalenceClass& fault = faults[i];
                        if (fault.intersects(packetClass)) {
```

```
                        vector<EquivalenceClass> diff = fault.subtract(packetClass);
                        for (auto&ec : diff) {
                                newFaults.push_back(ec);
                        }
                }
                else {
                        newFaults.push_back(fault);
                }

        }
        faults = newFaults;
        faults.push_back(packetClass);

        return false;
}

visited.insert(currentLocation);
path.push_back(currentLocation);

if(graph->links.find(currentLocation) == graph->links.end())
{
        // Found a black hole.
        fprintf(fp, "\n");
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet cla
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_st
        vector<EquivalenceClass> newFaults;


        for(unsigned int i = 0; i < faults.size(); i++) {
                EquivalenceClass& fault = faults[i];
                if (fault.intersects(packetClass)) {
                        vector<EquivalenceClass> diff = fault.subtract(packetClass);
                        for (auto&ec : diff) {
                                newFaults.push_back(ec);
                        }
                }
                else {
                        newFaults.push_back(fault);
                }

        }
        faults = newFaults;
        faults.push_back(packetClass);
        return false;
}

if(graph->links[currentLocation].empty() == true)
{
        // Found a black hole.
        fprintf(fp, "\n");
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet cla
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_st
        vector<EquivalenceClass> newFaults;

        for(unsigned int i = 0; i < faults.size(); i++) {
```

```
                EquivalenceClass& fault = faults[i];
                if (fault.intersects(packetClass)) {
                        vector<EquivalenceClass> diff = fault.subtract(packetClass);
                        for (auto&ec : diff) {
                                newFaults.push_back(ec);
                        }
                }
                else {
                        newFaults.push_back(fault);
                }

        }
        faults = newFaults;
        faults.push_back(packetClass);


        return false;
}

graph->links[currentLocation].sort(compareForwardingLink);

const list< ForwardingLink >& linkList = graph->links[currentLocation];
list< ForwardingLink >::const_iterator itr = linkList.begin();
// input_port as a filter
if(lastHop.compare("NULL") == 0 || itr->rule.in_port == 0){
        // do nothing
}
else{
        while(itr != linkList.end()){
                string connected_hop = network.getNextHopIpAddress(currentLocation, itr->rule.in_port);
                if(connected_hop.compare(lastHop) == 0) break;
                itr++;
        }
}

if(itr == linkList.end()){
        // Found a black hole.
        fprintf(fp, "\n");
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet cla
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_st

        vector<EquivalenceClass> newFaults;

        for(unsigned int i = 0; i < faults.size(); i++) {
                EquivalenceClass& fault = faults[i];
                if (fault.intersects(packetClass)) {
                        vector<EquivalenceClass> diff = fault.subtract(packetClass);
                        for (auto&ec : diff) {
                                newFaults.push_back(ec);
                        }
                }
                else {
                        newFaults.push_back(fault);
                }
```

```
                }
                faults = newFaults;
                faults.push_back(packetClass);

                return false;
        }

        if(itr->isGateway == true)
        {
                // Destination reachable.
                // fprintf(fp, "[VeriFlow::traverseForwardingGraph] Destination reachable.\n");
                fprintf(fp, "\n");
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] The following packet class reached destination
                fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_st
                vector<EquivalenceClass> newFaults;
                for(unsigned int i = 0; i < faults.size(); i++) {
                        EquivalenceClass& fault = faults[i];
                        if (fault.intersects(packetClass)) {
                                fprintf(stderr, "Removing fault!\n");
                                vector<EquivalenceClass> diff = fault.subtract(packetClass);
                                for (auto&ec : diff) {
                                        newFaults.push_back(ec);
                                }
                        }
                        else {
                                newFaults.push_back(fault);
                        }

                }
                faults = newFaults;

                return true;
        }
        else
        {
                // Move to the next location.
                // fprintf(fp, "[VeriFlow::traverseForwardingGraph] Moving to node %s.\n", itr->rule.nextHop.c_s

                if(itr->rule.nextHop.compare("") == 0)
                {
                        // This rule is a packet filter. It drops packets.
                        /* fprintf(fp, "\n");
                        fprintf(fp, "[VeriFlow::traverseForwardingGraph] The following packet class is dropped b
                        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toStrin
                }

                return this->traverseForwardingGraph(packetClass, graph, itr->rule.nextHop, currentLocation, vis
        }
}
```