

一、实验目的

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

二、实验过程

2.1 热身实验部分

2.1.1 观察转发环路

步骤一：启动拓扑和控制程序。

```
nakano9331@nakano9331-VirtualBox: ~/sdn-lab4
*** Starting controller
c0
*** Starting 8 switches
s1 (10.00Mbit 2ms delay) (10.00Mbit 1ms delay) s2 (10.00Mbit 2ms delay) (10.00Mbit 1ms delay) s3 (10.00Mbit 43ms delay) (10.00Mbit 4ms delay) (10.00Mbit 7ms delay) s4 (10.00Mbit 7ms delay) (10.00Mbit 46ms delay) s5 (10.00Mbit 4ms delay) (10.00Mbit 34ms delay) s6 (10.00Mbit 1ms delay) (10.00Mbit 43ms delay) s7 (10.00Mbit 1ms delay) (10.00Mbit 46ms delay) s8 (10.00Mbit 34ms delay) ... (10.00Mbit 2ms delay) (10.00Mbit 1ms delay) (10.00Mbit 2ms delay) (10.00Mbit 1ms delay) (10.00Mbit 7ms delay) (10.00Mbit 4ms delay) (10.00Mbit 43ms delay) (10.00Mbit 7ms delay) (10.00Mbit 46ms delay) (10.00Mbit 4ms delay) (10.00Mbit 34ms delay) (10.00Mbit 1ms delay) (10.00Mbit 43ms delay) (10.00Mbit 1ms delay) (10.00Mbit 46ms delay)
(10.00Mbit 34ms delay)
illinois illinois-eth0:s3-eth1
psu psu-eth0:s8-eth1
purdue purdue-eth0:s5-eth1
ucla1 ucla1-eth0:s1-eth1
ucla2 ucla2-eth0:s6-eth1
usc1 usc1-eth0:s2-eth1
usc2 usc2-eth0:s7-eth1
wisconsin wisconsin-eth0:s4-eth1
*** Starting CLI:
mininet>

nakano9331@nakano9331-VirtualBox: ~/sdn-lab4$ TOPO=simple.txt CONFIG=simple.conf
g.json uv run ryu-manager ryu.app.ofctl_rest as_switch.py
loading app ryu.app.ofctl_rest
loading app as_switch.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgl
creating context network_awareness
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app as_switch.py of RoutingSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
(3408) wsgl starting up on http://0.0.0.0:8080
```

步骤二：在拓扑中 ucla2 ping purdue 建立连接。

```
mininet> ucla2 ping purdue
PING 10.10.0.3 (10.10.0.3) 56(84) bytes of data.
64 字节, 来自 10.10.0.3: icmp_seq=1 ttl=64 时间=139 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=2 ttl=64 时间=106 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=3 ttl=64 时间=110 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=4 ttl=64 时间=142 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=5 ttl=64 时间=138 毫秒
^C
--- 10.10.0.3 ping 统计 ---
已发送 5 个包, 已接收 5 个包, 0% 包丢失, 耗时 4009 毫秒
rtt min/avg/max/mdev = 106.168/127.195/142.433/15.568 ms
```

此时观察通讯路径, 控制器中显示为 6 -> s6:3 -> s3:3 -> s5:1 -> 10.10.0.3。

```
path: 6 -> 10.10.0.3
6 -> s6:3 -> s3:3 -> s5:1 -> 10.10.0.3
```

步骤三：执行 `uv run ./gen_loop.py` 下发新路径。

```
nakano9331@nakano9331-VirtualBox: ~/sdn-lab4$ uv run ./gen_loop.py
nakano9331@nakano9331-VirtualBox: ~/sdn-lab4$
```

执行 gen_loop.py 下发从 illinois 途经 wisconsin 到达 ucla2 的路径之后，尝试 ucla2 ping purdue 失败，具体情况如下所示。

```
mininet> ucla2 ping purdue
PING 10.10.0.3 (10.10.0.3) 56(84) bytes of data.
```

步骤四：查看路径上某一个交换机，如 illinois 的流表。

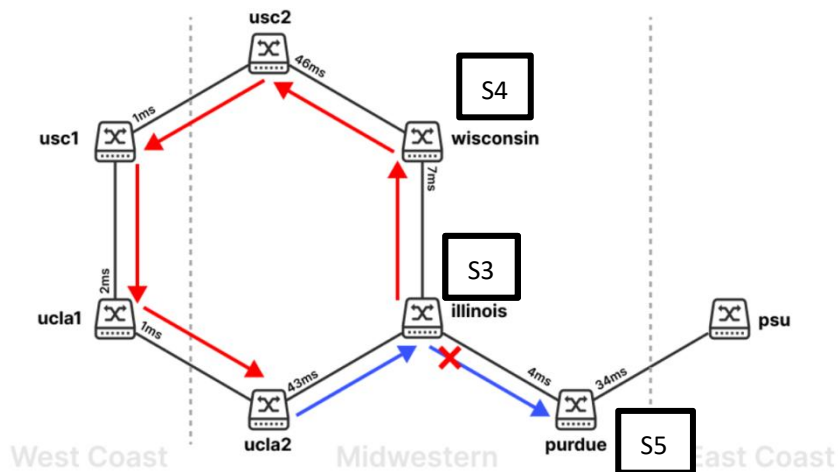
执行 sudo ovs-ofctl dump-flows s3 发现匹配某一条流表的数据包数目异常增加，高达 19303 个 packet，具体情况如下所示，说明了路由环路的存在。

```
nakano9331@nakano9331-VirtualBox:~/sdn-lab4$ sudo ovs-ofctl dump-flows s3
cookie=0x0, duration=115.516s, table=0, n_packets=19303, n_bytes=1891694, priority=10,ip,nw_dst=10.10.0.16
actions=output:"s3-eth4"
cookie=0x0, duration=350.969s, table=0, n_packets=5, n_bytes=490, priority=5,ip,nw_src=10.10.0.4,nw_dst=10.1
0.0.3 actions=output:"s3-eth3"
cookie=0x0, duration=350.910s, table=0, n_packets=5, n_bytes=490, priority=5,ip,nw_src=10.10.0.3,nw_dst=10.1
0.0.4 actions=output:"s3-eth2"
cookie=0x0, duration=418.709s, table=0, n_packets=31, n_bytes=2725, priority=0 actions=CONTROLLER:65509
```

使用 Wireshark 观察 S2-eth2 端口，发现大量源 IP 地址为 10.0.0.4，目的 IP 地址为 10.0.0.3 的 ICMP 请求报文，这也从另一方面说明了路由环路的存在。

No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=20/5120, ttl=64 (no respo...
2	0.000000374	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=19/4864, ttl=64 (no respo...
3	0.000000413	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=59/15104, ttl=64 (no respo...
4	0.000000449	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=53/13568, ttl=64 (no respo...
5	0.000000485	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=45/11520, ttl=64 (no respo...
6	0.000000519	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=34/8704, ttl=64 (no respo...
7	0.000000554	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=49/12544, ttl=64 (no respo...
8	0.000000589	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=49/12544, ttl=64 (no respo...
9	0.000000624	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=46/11776, ttl=64 (no respo...
10	0.000000673	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=9/2304, ttl=64 (no respo...
11	0.000010089	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=7/1792, ttl=64 (no respo...
12	0.000010431	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0ea5, seq=13/3328, ttl=64 (no respo...
13	0.000010771	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=44/11264, ttl=64 (no respo...
14	0.000011110	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=41/10496, ttl=64 (no respo...
15	0.000011581	10.10.0.4	10.10.0.3	ICMP	98	Echo (ping) request id=0x0d6e, seq=37/9472, ttl=64 (no respo...

路径的重新下发导致了形如 S3 处的流表改变，导致 ILLINOIS 到 PURDUE 的数据包转发错误，即 controller 控制全局 Switch 的 FlowTable，最开声明使 S3 数据包发往 S5，随后加入新链路又使得 S3 发往 S4，故形成了环路。



2.1.2 使用 VeriFlow 验证环路问题

步骤一：编译 VeriFlow。

执行命令 make -C veriflow -j\$(nproc) 进行编译，其中，-j\$(nproc) 为指

定编译线程数，可以不加。

步骤二：在自定义端口开启远程控制器，运行 AS 控制器程序，执行命令后结果如下所示。

```
nakano9331@nakano9331-VirtualBox:~/sdn-Lab4$ TOPO=simple.txt CONFIG=simple.config.json uv run ryu-manager ryu.app.ofctl_rest as_swl
tch.py --ofp-tcp-listen-port 1024
loading app ryu.app.ofctl_rest
loading app as_switch.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
creating context dpset
creating context wsgi
instantiating app None of NetworkAwareness
creating context network_awareness
instantiating app ryu.app.ofctl_rest of RestStatsApi
instantiating app as_switch.py of RoutingSwitch
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
(4217) wsgi starting up on http://0.0.0.0:8080
```

步骤三：执行 `./VeriFlow 6633 127.0.0.1 1024 simple.txt veriflow.log` 命令运行 VeriFlow 的 proxy 模式。

步骤四：启动拓扑，并在拓扑中 ucla2 ping purdue 建立连接。

步骤五：再次执行 `uv run ./gen_loop.py` 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。

首先观察到，在 ucla2 ping purdue 建立连接后，生成转发环路的过程与先前操作过程相同。在下发从 UTAH 途经 TINKER 到达 ILLINOIS 的路径后，log 文件中 VeriFlow 记录的信息如下所示。

```
mininet> ucla2 ping purdue
PING 10.10.0.3 (10.10.0.3) 56(84) bytes of data.
64 字节, 来自 10.10.0.3: icmp_seq=1 ttl=64 时间=163 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=2 ttl=64 时间=120 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=3 ttl=64 时间=104 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=4 ttl=64 时间=102 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=5 ttl=64 时间=98.9 毫秒
64 字节, 来自 10.10.0.3: icmp_seq=6 ttl=64 时间=102 毫秒
^C
--- 10.10.0.3 ping 统计 ---
已发送 6 个包, 已接收 6 个包, 0% 包丢失, 耗时 5094 毫秒
rtt min/avg/max/mdev = 98.861/114.976/163.288/22.660 ms
```

由于控制器的工作特性限制，其无法在同一时间内同时下发多个流表项。在这种情况下，VeriFlow 对特定流表项进行检查时，可能会出现以下情况。

```
[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as
current location (20.0.0.006.6) not found in the graph.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
(0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00,
281474976710655-ff:ff:ff:ff:ff:ff), nw_src (168427523-10.10.0.3, 168427523-10.10.0.3),
nw_dst (168427520-10.10.0.0, 168427523-10.10.0.3), Field 0 (0, 65535), Field 1 (0,
281474976710655), Field 2 (0, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095),
Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (168427523, 168427523), Field
9 (168427520, 168427523), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field
13 (0, 65535)
```

当数据包依据当前流表项匹配规则被转发至下一个交换机后，却发现下一个交换机中尚未部署用于处理该数据包的相应流表项。面对这种状况，VeriFlow 会对黑洞信息进行记录。

```
[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node
20.0.0.006.6.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
(0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00,
281474976710655-ff:ff:ff:ff:ff:ff), nw_src (168427523-10.10.0.3, 168427523-10.10.0.3),
nw_dst (168427524-10.10.0.4, 168427524-10.10.0.4), Field 0 (0, 65535), Field 1 (0,
281474976710655), Field 2 (0, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095),
Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (168427523, 168427523), Field
9 (168427524, 168427524), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field
13 (0, 65535)
```

当某条流表项具备使数据包顺利转发至目的主机的能力时，这意味着目的主机可以通过该流表项实现可达。此时，VeriFlow 会将这一目的主机可达信息进行记录，为网络拓扑和流量管理提供参考依据。

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node
20.0.0.006.6.
[VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
(0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00,
281474976710655-ff:ff:ff:ff:ff:ff), nw_src (0-0.0.0.0, 168427522-10.10.0.2), nw_dst
(168427520-10.10.0.0, 168493055-10.10.255.255), Field 0 (0, 65535), Field 1 (0,
281474976710655), Field 2 (0, 281474976710655), Field 3 (2048, 2048), Field 4 (0, 4095),
Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (0, 168427522), Field 9
(168427520, 168493055), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13
(0, 65535)
```

在新流表项的下发过程中，VeriFlow 会对新流表项可能对网络路由产生的影响进行实时监测。一旦发现新流表项的引入会导致路由环路产生，VeriFlow 在日志文件中记录该路由环路信息。

2.1.3 VeriFlow 内容修改

步骤一：输出每次影响 EC 的数量。VeriFlow::verifyRule() 为执行 VeriFlow 核心算法的函数，包括对等价类的划分、转发图的构造与不变量的验证。函数中变量 ecCount 为 EC 数目，将等价类定向输出到 fp，即 log_file.txt 文件中。

```
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). Terminating process.\n");
    exit(1);
}
else
{
    fprintf(stdout, "\n");
    fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
    fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
}
```

步骤二：输出环路路径详细信息。在 VeriFlow::traverseForwardingGraph() 函数中，系统将对特定等价类的转发图展开遍历操作，以此验证网络中是否存在环路或黑洞问题。

在该函数执行过程中，visited 变量被用于记录已经遍历过的节点。当检测到当前节点已存在于 visited 记录中意味着环路产生。利用哈希技术实现元素的无序存储，这种方式能够在保证数据存储高效性的同时，快速完成元素的查找判断。此处选择新增一个 vector<string> 类型的变量 loop_path，在遍历过程中实时记录路径信息，从而完整保存环路的具体走向。

```
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 node %s\n", currentLocation.c_str());
    fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());
    fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");
    for(unsigned int i = 0; i < loop_path.size(); i++){
        fprintf(fp, "%s -> ", loop_path[i].c_str());
    }
    fprintf(fp, "%s\n", currentLocation.c_str());
    for(unsigned int i = 0; i < faults.size(); i++) {
        if (packetClass.subsumes(faults[i])) {
            faults.erase(faults.begin() + i);
            i--;
        }
    }
    faults.push_back(packetClass);
    return false;
}

visited.insert(currentLocation);
loop_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 class as current location (%s) not found in the graph.\n", currentLocation.c_str());
    fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());
    for(unsigned int i = 0; i < faults.size(); i++) {
        if (packetClass.subsumes(faults[i])) {
            faults.erase(faults.begin() + i);
            i--;
        }
    }
    faults.push_back(packetClass);
    return false;
}
```


步骤三：进一步打印出环路对应的 EC 的相关信息。

在 EquivalenceClass 类中修改函数 toString() 函数，将原来的输出，改成输出标准 TCP/IP 五元组，即为(IP 源地址, IP 目的地址, 网络层协议类型, 传输层源端口号, 传输层目的端口号)。

```
char buffer[1024];
memset(buffer,0,sizeof(buffer));
string retVal = buffer;

sprintf(buffer,"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),nw_src(%lu-%lu),nw_dst(%lu-%lu)",
this->lowerBound[NW_PROTO],
this->upperBound[NW_PROTO],
this->lowerBound[TP_SRC],
this->upperBound[TP_SRC],
this->lowerBound[TP_DST],
this->upperBound[TP_DST]);

retVal += buffer;
return retVal;
```

重复上一部分各个步骤，在 log 文件中观察 VeriFlow 检测到的环路信息，最终结果如下所示。

```
155 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as current location (20.0.0.006.6) not found in the graph.
156 [VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.10.0.5-255.255.255.255),nw_dst(10.10.0.0-10.10.255.255),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)
157 [VeriFlow::verifyRule] ecCount: 8
158
159 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.006.6.
160 [VeriFlow::traverseForwardingGraph] PacketClass: nw_src(0.0.0.0-10.10.0.2),nw_dst(10.10.0.0-10.10.255.255),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)
161 [VeriFlow::traverseForwardingGraph] Loop path is:
162 20.0.0.006.6 -> 20.0.0.003.3 -> 20.0.0.004.4 -> 20.0.0.007.7 -> 20.0.0.002.2 -> 20.0.0.001.1 -> 20.0.0.006.6
163
164 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.006.6.
165 [VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.10.0.3-10.10.0.3),nw_dst(10.10.0.0-10.10.0.3),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)
166 [VeriFlow::traverseForwardingGraph] Loop path is:
167 20.0.0.006.6 -> 20.0.0.003.3 -> 20.0.0.004.4 -> 20.0.0.007.7 -> 20.0.0.002.2 -> 20.0.0.001.1 -> 20.0.0.006.6
168
169 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.006.6.
170 [VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.10.0.3-10.10.0.3),nw_dst(10.10.0.4-10.10.0.4),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)
171 [VeriFlow::traverseForwardingGraph] Loop path is:
172 20.0.0.006.6 -> 20.0.0.003.3 -> 20.0.0.004.4 -> 20.0.0.007.7 -> 20.0.0.002.2 -> 20.0.0.001.1 -> 20.0.0.006.6
173
```

2.2 基础实验部分

2.1.1 建立转发黑洞

步骤一：在自定义端口开启远程控制器，运行 AS 控制器程序。

步骤二：运行 VeriFlow 的 proxy 模式并启动拓扑。

步骤三：建立转发路径，观察到此时，两者可以正常 ping 通，但是 VeriFlow 提示出现了黑洞。

```
mininet> ucla1 ping illinois
PING 10.10.0.1 (10.10.0.1) 56(84) bytes of data.
64 字节, 来自 10.10.0.1: icmp_seq=1 ttl=64 时间=172 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=2 ttl=64 时间=99.8 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=3 ttl=64 时间=90.0 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=4 ttl=64 时间=95.3 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=5 ttl=64 时间=93.6 毫秒
^C
--- 10.10.0.1 ping 统计 ---
已发送 5 个包, 已接收 5 个包, 0% 包丢失, 耗时 4017 毫秒
rtt min/avg/max/mdev = 89.967/110.212/172.363/31.237 ms
```

```
[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as current location (20.0.0.006.6) not found in the graph.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.12.0.0-10.12.255.255),nw_dst(10.10.0.0-10.10.255.255),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)
[VeriFlow::verifyRule] Network Broken!
[VeriFlow::verifyRule] ecCount: 1
```

```
[VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as current location (20.0.0.006.6) not found in the graph.  
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.10.0.0-10.10.127.255),nw_dst(10.12.0.0-10.12.255.255),nw_proto(0-255),nw_src(0-65535),nw_dst(0-65535)  
[VeriFlow::verifyRule] ecCount: 1
```

通过观察 log 记录，发现节点编号 20.0.0.006.6，即 S6 交换机未正确建立转发路径，在该交换机上，找不到对应的转发条目来处理指定的包。ucla1 发包到 illinois 时，走到 ucla2 处，不知道下一跳方向。而 illinois 发包回 ucla1 时，走到 ucla2，也不知道怎么转发回去了。

在 fix_path.py 中，通过 send_flow_mod 函数添加了流表项。如向 S6 下发规则，当接收到目的 IP 为 10.10.0.0/16 的数据包时，将其从端口 3 转发出去。对于 S3 和 S1 则转发到端口 1 对应的主机。通过这一系列流表项的添加，为之前等价类的数据包在网络中指定了明确的转发路径。

然而，执行后 VeriFlow 发现的网络错误依旧存在。

```
[VeriFlow::verifyRule] ecCount: 11  
faults size: 12  
  
[VeriFlow::verifyRule] ecCount: 11  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
Removing fault!  
faults size: 2  
  
[VeriFlow::verifyRule] ecCount: 11  
faults size: 2
```

2.1.2 修复 Fault 计算

为了处理上述错误计算，修改 VeriFlow.cpp 中的 traverseForwardingGraph 函数，对转发图进行遍历，以此来检查数据包类在图中的转发状况，并且识别诸如环路、黑洞等问题。保留了原有对输入参数的检查、对已访问节点的检查、对当前节点是否存在出边的检查等关键步骤。对当前节点的出边列表以规则 compareForwardingLink 进行排序。根据输入端口过滤出边的逻辑，以防止数据包反向转发。

2.1.2.1 VeriFlow.cpp 修改部分

修改思路为调用 updateFaults 函数更新故障列表，该函数会对故障列表中的每个故障与当前数据包类求交集和差集，并且移除空的故障。此外，使用 isPacketClassInFaults 函数检查当前数据包类是否已存在于故障列表中。

具体部分见源代码处。

2.1.2.2 EquivalenceClass.cpp 修改部分

在 EquivalenceClass.cpp 中加入定义 EquivalenceClass 类的三个成员函数，分别是 intersection、difference 和 isEmpty。

①Intersection 用于计算当前两个对象的交集。首先初始化新的上下界数组，长度为 ALL_FIELD_INDEX_END_MARKER 的数组并将其命名为 newLowerBound 和 newUpperBound，用于存储交集的上下界。针对每个字段，取当前对象和 other 对象对应字段的下界的最大值作为新的下界，上界的最小值作为新的上界。若所有字段的上下界都为 0，则认为交集为空。此时返回一个空对象；否则，使用新的上下界数组创建一个新的 EquivalenceClass 对象并返回。

②difference 用于计算当前两个对象的差集。首先初始化结果部分向量，创建一个 std::vector<EquivalenceClass> 类型的 resultParts，用于存储差集的各个部分。对于每个字段，若 other 的上下界合法，则检查当前对象的上下界与 other 的上下界的关系，若存在差值部分，则创建新的 EquivalenceClass 对象并添加到 resultParts 中。随后对 resultParts 中所有部分进行合并，通过多次遍历所有字段，将重叠部分合并成一个更大的等价类。

若 resultParts 不为空，将所有部分合并成一个最终的对象并返回；否则，返回一个空的 EquivalenceClass 对象。

③isEmpty 用于判断当前 EquivalenceClass 对象是否为空。遍历所有字段，若存在某个字段的上下界不为 0，则返回 false；否则，返回 true。

各个部分修改内容，详见源代码处。

2.1.2.3 实验结果

uclal ping illinois 之前，运行 illinois ping wisconsin 如下所示。

```
nakano9331@nakano9331-VirtualBox: ~/sdn-lab4
[proxyCommunicationThreadFunction] Connection closed.
[proxyCommunicationThreadFunction] Connection closed.
[VeriFlow] [5] Accepted new connection from 127.0.0.1 at port 58596
[VeriFlow] [6] Waiting for new connection...
[VeriFlow] [6] Accepted new connection from 127.0.0.1 at port 58604
[VeriFlow] [7] Waiting for new connection...
[VeriFlow] [7] Accepted new connection from 127.0.0.1 at port 58612
[VeriFlow] [8] Waiting for new connection...
[VeriFlow] [8] Accepted new connection from 127.0.0.1 at port 58624
[VeriFlow] [9] Waiting for new connection...
[VeriFlow] [9] Accepted new connection from 127.0.0.1 at port 58640
[VeriFlow] [10] Waiting for new connection...
[VeriFlow] [10] Accepted new connection from 127.0.0.1 at port 58656
[VeriFlow] [11] Waiting for new connection...
[VeriFlow] [11] Accepted new connection from 127.0.0.1 at port 58664
[VeriFlow] [12] Waiting for new connection...
[VeriFlow] [12] Accepted new connection from 127.0.0.1 at port 58670
[VeriFlow] [13] Waiting for new connection...
faults size: 1
faults size: 0
faults size: 1
faults size: 0
faults size: 0

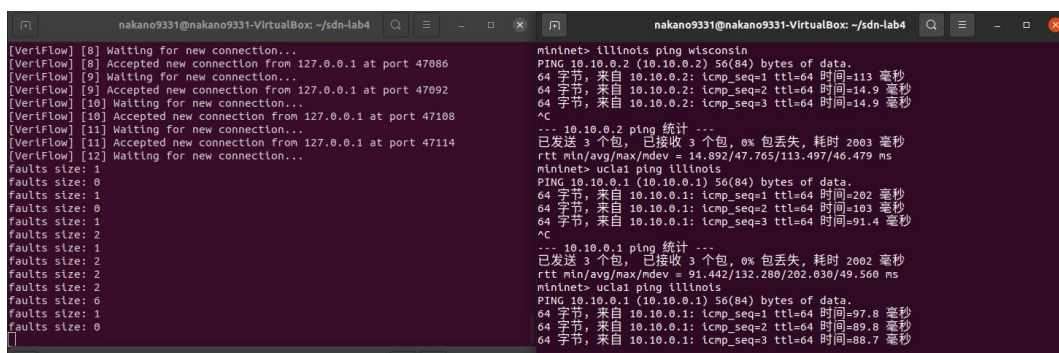
delay (10.00Mbit 1ms delay) (10.00Mbit 2ms delay) (10.00Mbit 1ms delay) (10.00Mbit 7ms delay) (10.00Mbit 4ms delay) (10.00Mbit 43ms delay) (10.00Mbit 7ms delay) (10.00Mbit 46ms delay) (10.00Mbit 4ms delay) (10.00Mbit 34ms delay) (10.00Mbit 1ms delay) (10.00Mbit 43ms delay) (10.00Mbit 1ms delay) (10.00Mbit 46ms delay)
illinois illinois-eth0:s3-eth1
psu psu-eth0:s8-eth1
purdue purdue-eth0:s5-eth1
uclal uclal-eth0:s1-eth1
ucla2 ucla2-eth0:s6-eth1
usc1 usc1-eth0:s2-eth1
usc2 usc2-eth0:s7-eth1
wisconsin wisconsin-eth0:s4-eth1
*** Starting CLI:
mininet> illinois ping wisconsin
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
64 字节, 来自 10.10.0.2: icmp_seq=1 ttl=64 时间=124 毫秒
64 字节, 来自 10.10.0.2: icmp_seq=2 ttl=64 时间=16.5 毫秒
64 字节, 来自 10.10.0.2: icmp_seq=3 ttl=64 时间=16.8 毫秒
^C
--- 10.10.0.2 ping 统计 ---
已发送 3 个包, 已接收 3 个包, 0% 包丢失, 耗时 2004 毫秒
rtt min/avg/max/mdev = 16.512/52.370/123.801/50.509 ms
mininet>
```

uclal ping illinois 之后，修复之前如下所示。

```
nakano9331@nakano9331-VirtualBox: ~/sdn-lab4
[VeriFlow] [7] Accepted new connection from 127.0.0.1 at port 58612
[VeriFlow] [8] Waiting for new connection...
[VeriFlow] [8] Accepted new connection from 127.0.0.1 at port 58624
[VeriFlow] [9] Waiting for new connection...
[VeriFlow] [9] Accepted new connection from 127.0.0.1 at port 58640
[VeriFlow] [10] Waiting for new connection...
[VeriFlow] [10] Accepted new connection from 127.0.0.1 at port 58656
[VeriFlow] [11] Waiting for new connection...
[VeriFlow] [11] Accepted new connection from 127.0.0.1 at port 58664
[VeriFlow] [12] Waiting for new connection...
[VeriFlow] [12] Accepted new connection from 127.0.0.1 at port 58670
[VeriFlow] [13] Waiting for new connection...
faults size: 1
faults size: 0
faults size: 1
faults size: 0
faults size: 1
faults size: 2
faults size: 2
faults size: 2
faults size: 2
faults size: 2
faults size: 2

uclal2 uclal2-eth0:s6-eth1
usc1 usc1-eth0:s2-eth1
usc2 usc2-eth0:s7-eth1
wisconsin wisconsin-eth0:s4-eth1
*** Starting CLI:
mininet> illinois ping wisconsin
PING 10.10.0.2 (10.10.0.2) 56(84) bytes of data.
64 字节, 来自 10.10.0.2: icmp_seq=1 ttl=64 时间=124 毫秒
64 字节, 来自 10.10.0.2: icmp_seq=2 ttl=64 时间=16.5 毫秒
64 字节, 来自 10.10.0.2: icmp_seq=3 ttl=64 时间=16.8 毫秒
^C
--- 10.10.0.2 ping 统计 ---
已发送 3 个包, 已接收 3 个包, 0% 包丢失, 耗时 2004 毫秒
rtt min/avg/max/mdev = 16.512/52.370/123.801/50.509 ms
mininet> uclal1 ping illinois
PING 10.10.0.1 (10.10.0.1) 56(84) bytes of data.
64 字节, 来自 10.10.0.1: icmp_seq=1 ttl=64 时间=223 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=2 ttl=64 时间=98.9 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=3 ttl=64 时间=98.9 毫秒
^C
--- 10.10.0.1 ping 统计 ---
已发送 3 个包, 已接收 3 个包, 0% 包丢失, 耗时 2249 毫秒
rtt min/avg/max/mdev = 98.850/140.200/222.840/58.435 ms
mininet>
```


修复之后如下所示。



The image shows two terminal windows. The left window displays logs from 'VeriFlow' showing connections from 127.0.0.1 at various ports (47086, 47092, 47108, 47114) and a series of 'faults size' updates (0, 1, 0, 1, 0, 2, 1, 2, 2, 6, 1, 0). The right window shows 'mininet> illinois ping wisconsin' and 'mininet> ucla1 ping illinois' commands, followed by detailed ping statistics for both paths, including packet counts, loss percentages, and round-trip times.

如图所示，通过处理新加入规则等价类和原有错误之间关系，成功将 Faults 计算问题修复。

2.3 拓展实验部分

本部分选择阅读 `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。

`as_switch.py` 主要功能是处理网络中的数据包，进行二层自学习和三层路由转发。它首先获取数据包的相关信息，如数据路径、交换机 ID、输入端口 `in_port` 等。然后解析数据包，获取以太网层、ARP 层和 IPv4 层的协议对象。接着进行二层自学习，将源 MAC 地址和输入端口的映射关系存储在 `dpid_mac_port` 中。最后根据数据包的类型，分别调用 `handle_arp` 或 `handle_ipv4` 方法进行处理。

`handle_arp` 当目标 MAC 地址在 `dpid_mac_port` 中有对应的输出端口，则直接将数据包从该端口发送出去。否则，将数据包发送到所有除源主机所在的端口以外，连接主机的交换机端口。

`handle_ipv4` 获取当前交换机的子网 `switch_net`，然后查找源 IP 地址所属的子网 `srcnet` 和目标 IP 地址所属的子网 `dstnet`。如果目标 IP 地址在当前交换机的子网内，则进行本地交换，`gateways` 设为 `None`；否则，查找目标 IP 地址所属子网的网关 `gateways`。如果无法识别源或目标子网，则无法转发数据包并返回。

`add_path` 是一个内部函数，用于添加从源到目标的路径。它首先根据路径 `route` 确定每个跳的输出端口，记录在 `port_path` 中。然后展示路径信息，并在每个路径节点上发送流表项，最后返回当前交换机的输出端口。

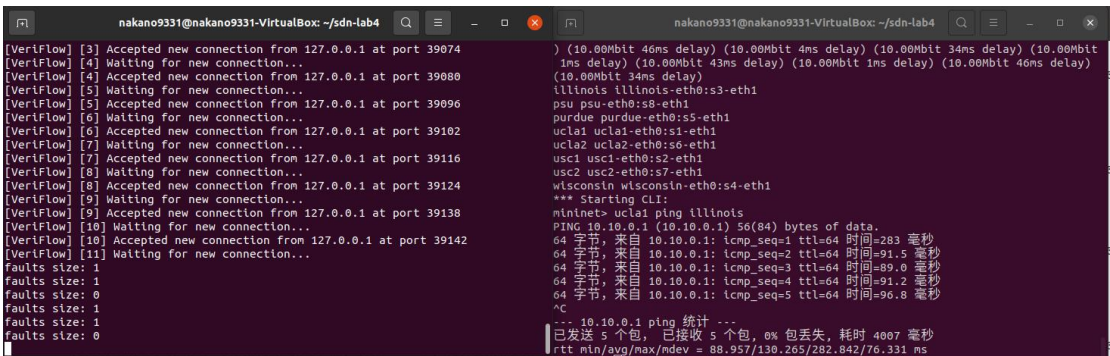
根据 `gateways` 是否为 `None` 来判断是进行 AS 内的数据包交换还是 AS 间的数据包交换。如果是 AS 内交换，获取从当前交换机到目标 IP 地址的最短路径 `dpid_path`，并调用 `add_path` 添加路径；如果是 AS 间交换，判断当前交换机是否为网关，如果是则发送到对等网关，否则找到最近的网关，获取到该网关的最短路径并添加路径。在安装路径后，根据获取的输出端口 `out_port`，构造 `OFPPacketOut` 消息，将数据包从该端口发送出去。

产生黑洞的原因在于跨 AS 后进入了 AS 内交换的逻辑中，开始计算从当前交换机到目标 IP 地址的最短路径 `dpid_path`，并调用 `add_path` 添加路径，是以准确的 IP 为目标下发流表，当遇到以 /16 的子网目标时，查询不到相应流

表导致出现黑洞，这种黑洞错误若是在以相同的源地址和目标地址进行通信时，虽然会出现报错，但不影响通信。但若是以相同子网，不同源地址或目标地址进行通信时，就将影响通信，对网络的实际使用造成影响。

为了解决这个问题，修改 `as_switch.py`，在 `add_path` 方法中，除了下发原流表外，还会根据源 IP 和目的 IP 计算出对应的 /16 子网范围，并添加更高优先级的流表。这一操作可以提高匹配效率，同时可能有助于处理一些特殊的网络需求或优化网络性能。计算 /16 子网范围的方法通过 `get_subnet_16` 方法实现的，该方法将 IP 地址的前两个字节保留，后两个字节设置为 0，并添加 /16 的子网掩码。且无论 `gateways` 的情况如何，都统一使用子网范围作为源 IP 和目的 IP 的参数传入。这样做使得代码逻辑更加一致和清晰，便于理解和维护。

可以看出实验结果如下所示，黑洞问题将不再出现。



The screenshot shows a terminal window with two panes. The top pane displays the output of a network simulation, showing various network events and ping statistics. The bottom pane shows the contents of a file named `veriflow.log`, which contains detailed logs of network operations, including packet forwarding and rule verification.

```
[VeriFlow] [3] Accepted new connection from 127.0.0.1 at port 39074
[VeriFlow] [4] Waiting for new connection...
[VeriFlow] [4] Accepted new connection from 127.0.0.1 at port 39080
[VeriFlow] [5] Waiting for new connection...
[VeriFlow] [5] Accepted new connection from 127.0.0.1 at port 39096
[VeriFlow] [6] Waiting for new connection...
[VeriFlow] [6] Accepted new connection from 127.0.0.1 at port 39102
[VeriFlow] [7] Waiting for new connection...
[VeriFlow] [7] Accepted new connection from 127.0.0.1 at port 39116
[VeriFlow] [8] Waiting for new connection...
[VeriFlow] [8] Accepted new connection from 127.0.0.1 at port 39124
[VeriFlow] [9] Waiting for new connection...
[VeriFlow] [9] Accepted new connection from 127.0.0.1 at port 39138
[VeriFlow] [10] Waiting for new connection...
[VeriFlow] [10] Accepted new connection from 127.0.0.1 at port 39142
[VeriFlow] [11] Waiting for new connection...
faults size: 1
faults size: 1
faults size: 0
faults size: 1
faults size: 1
faults size: 0
faults size: 0

) (10.00Mbit 46ms delay) (10.00Mbit 4ms delay) (10.00Mbit 34ms delay) (10.00Mbit
1ms delay) (10.00Mbit 43ms delay) (10.00Mbit 1ms delay) (10.00Mbit 46ms delay)
(10.00Mbit 34ms delay)
illinois illinois-eth0:s3-eth1
psu psu-eth0:s8-eth1
purdue purdue-eth0:s5-eth1
ucla1 ucla1-eth0:s1-eth1
ucla2 ucla2-eth0:s6-eth1
usc1 usc1-eth0:s2-eth1
usc2 usc2-eth0:s7-eth1
wisconsin wisconsin-eth0:s4-eth1
*** Starting CLI:
ninet- ucla1 ping illinois
PING 10.10.0.1 (10.10.0.1) 56(84) bytes of data.
64 字节, 来自 10.10.0.1: icmp_seq=1 ttl=64 时间=283 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=2 ttl=64 时间=91.5 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=3 ttl=64 时间=89.0 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=4 ttl=64 时间=91.2 毫秒
64 字节, 来自 10.10.0.1: icmp_seq=5 ttl=64 时间=96.8 毫秒
^C
--- 10.10.0.1 ping 统计 ---
已发送 5 个包, 已接收 5 个包, 0% 包丢失, 耗时 4007 毫秒
rtt min/avg/max/mdev = 88.957/130.265/282.842/76.331 ms

2 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for packet class at node 20.0.0.006.6.
3 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108558592-10.12.0.0, 108624127-10.12.255.255), nw_dst (108427520-10.10.0.0, 108493055-10.10.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108558592, 108624127), Field 9 (108427520, 108493055), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
4 [VeriFlow::verifyRule] Network Broken!
5
6 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for packet class at node 20.0.0.003.3.
7 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108558592-10.12.0.0, 108624127-10.12.255.255), nw_dst (108427520-10.10.0.0, 108493055-10.10.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108558592, 108624127), Field 9 (108427520, 108493055), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
8
9 [VeriFlow::traverseForwardingGraph] Packet class reached destination at node 20.0.0.003.3.
10 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108558592-10.12.0.0, 108624127-10.12.255.255), nw_dst (108427520-10.10.0.0, 108493055-10.10.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108558592, 108624127), Field 9 (108427520, 108493055), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
11 [VeriFlow::verifyRule] Network Fixed!
12
13 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for packet class at node 20.0.0.006.6.
14 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108558592-10.12.0.0, 108624127-10.12.255.255), nw_dst (108427520-10.10.0.0, 108493055-10.10.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108427520, 108493055), Field 9 (108558592, 108624127), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
15 [VeriFlow::verifyRule] Network Broken!
16
17 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for packet class at node 20.0.0.001.1.
18 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108427520-10.10.0.0, 108493055-10.10.255.255), nw_dst (108558592-10.12.0.0, 108624127-10.12.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108427520, 108493055), Field 9 (108558592, 108624127), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
19
20 [VeriFlow::traverseForwardingGraph] Packet class reached destination at node 20.0.0.001.1.
21 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), dl_dst (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
nw_src (108427520-10.10.0.0, 108493055-10.10.255.255), nw_dst (108558592-10.12.0.0, 108624127-10.12.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 281474976710655), Field 3
(2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (108427520, 108493055), Field 9 (108558592, 108624127), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0,
65535), Field 13 (0, 65535)
22 [VeriFlow::verifyRule] Network Fixed!
```

三、源代码

Veriflow.cpp 修改部分

```
bool VeriFlow::traverseForwardingGraph(const EquivalenceClass&
packetClass, ForwardingGraph* graph, const string& currentLocation,
const string& lastHop, unordered_set<string> visited, FILE* fp)
{
    if (graph == NULL) {
```

```

        return true;
    }

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

    if (visited.find(currentLocation) != visited.end()) {
        fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Found a LOOP
for packet class at node %s.\n", currentLocation.c_str());
        fprintf(fp,
                "[VeriFlow::traverseForwardingGraph]
PacketClass: %s\n", packetClass.toString().c_str());
        updateFaults(packetClass, fp);
        if (!isPacketClassInFaults(packetClass)) {
            faults.push_back(packetClass);
        }
        return false;
    }

    visited.insert(currentLocation);

    if (graph->links.find(currentLocation) == graph->links.end()) {
        fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Found a BLACK
HOLE for packet class at node %s.\n", currentLocation.c_str());
        fprintf(fp,
                "[VeriFlow::traverseForwardingGraph]
PacketClass: %s\n", packetClass.toString().c_str());
        updateFaults(packetClass, fp);
        if (!isPacketClassInFaults(packetClass)) {
            faults.push_back(packetClass);
        }
        return false;
    }

    if (graph->links[currentLocation].empty()) {
        fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Found a BLACK
HOLE for packet class at node %s.\n", currentLocation.c_str());
        fprintf(fp,
                "[VeriFlow::traverseForwardingGraph]
PacketClass: %s\n", packetClass.toString().c_str());
        updateFaults(packetClass, fp);
        if (!isPacketClassInFaults(packetClass)) {
            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();

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()) {
    fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Found a BLACK
HOLE for packet class at node %s.\n", currentLocation.c_str());
    fprintf(fp,
            "[VeriFlow::traverseForwardingGraph]
PacketClass: %s\n", packetClass.toString().c_str());
    updateFaults(packetClass, fp);
    if (!isPacketClassInFaults(packetClass)) {
        faults.push_back(packetClass);
    }
    return false;
}

if (itr->isGateway == true) {
    fprintf(fp, "\n[VeriFlow::traverseForwardingGraph] Packet class
reached destination at node %s.\n", currentLocation.c_str());
    fprintf(fp,
            "[VeriFlow::traverseForwardingGraph]
PacketClass: %s\n", packetClass.toString().c_str());
    updateFaults(packetClass, fp);
    return true;
}
else {
    return this->traverseForwardingGraph(packetClass, graph,
itr->rule.nextHop, currentLocation, visited, fp);
}
}

```



```

void VeriFlow::updateFaults(const EquivalenceClass& packetClass, FILE*
fp) {
    //std::cout << "--- Starting updateFaults ---" << std::endl;
    //std::cout << "Before update, faults size: " << faults.size() <<
std::endl;
    int index = 0;
    for (auto it = faults.begin(); it != faults.end(); ) {
        //std::cout << "\nProcessing fault at index " << index << ":"
<< std::endl;
        //std::cout << "Fault before update: ";
        //it->printInfo("");

        EquivalenceClass intersection =
it->intersection(packetClass);
        if (!intersection.isEmpty()) {
            //std::cout << "Intersection found:" << std::endl;
            //intersection.printInfo("Intersection");

            EquivalenceClass diff = it->difference(intersection);
            //std::cout << "Difference result:" << std::endl;
            //diff.printInfo("Difference");

            *it = diff;
            //std::cout << "Fault after update: ";
            //it->printInfo("");

            if (it->isEmpty()) {
                //std::cout << "Fault is empty, removing it from faults
list." << std::endl;
                it = faults.erase(it);
            } else {
                //std::cout << "Fault is not empty, keeping it in faults
list." << std::endl;
                ++it;
            }
        } else {
            //std::cout << "No intersection, keeping the fault in faults
list." << std::endl;
            ++it;
        }
        ++index;
    }
    //std::cout << "After update, faults size: " << faults.size() <<

```

```

std::endl;
    //std::cout << "--- Ending updateFaults ---" << std::endl;
}

bool VeriFlow::isPacketClassInFaults(const EquivalenceClass&
packetClass)
{
    for (const auto& fault : faults) {
        if (fault.equals(packetClass)) {
            return true;
        }
    }
    return false;
}

```

EquivalenceClass.cpp 修改部分

```

EquivalenceClass EquivalenceClass::intersection(const
EquivalenceClass& other) const {
    uint64_t newLowerBound[ALL_FIELD_INDEX_END_MARKER];
    uint64_t newUpperBound[ALL_FIELD_INDEX_END_MARKER];

    bool allEmpty = true;
    for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
        newLowerBound[i] = std::max(this->lowerBound[i],
other.lowerBound[i]);
        newUpperBound[i] = std::min(this->upperBound[i],
other.upperBound[i]);

        //if (newLowerBound[i] > newUpperBound[i]) {
            //uint64_t temp = newLowerBound[i];
            //newLowerBound[i] = newUpperBound[i];
            //newUpperBound[i] = temp;
        //}

        if (newLowerBound[i] == 0 && newUpperBound[i] == 0) {
            continue;
        } else {
            allEmpty = false;
        }
    }

    if (allEmpty) {
        return EquivalenceClass();
    }
}

```

```

    }

    return EquivalenceClass(newLowerBound, newUpperBound);
}

EquivalenceClass EquivalenceClass::difference(const EquivalenceClass&
other) const {
    std::vector<EquivalenceClass> resultParts;
    const EquivalenceClass& intersectionResult = other;
    for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
        if (intersectionResult.lowerBound[i] <=
intersectionResult.upperBound[i]) {

            if (this->lowerBound[i] < intersectionResult.lowerBound[i]) {
                EquivalenceClass part = *this;
                part.upperBound[i] = intersectionResult.lowerBound[i] - 1;
                resultParts.push_back(part);
            }

            if (this->upperBound[i] > intersectionResult.upperBound[i]) {
                EquivalenceClass part = *this;
                part.lowerBound[i] = intersectionResult.upperBound[i] + 1;
                resultParts.push_back(part);
            }
        }
    }

    if (!resultParts.empty()) {

        for (int field = 0; field < ALL_FIELD_INDEX_END_MARKER; field++)
        {
            std::vector<EquivalenceClass> newResultParts;
            while (!resultParts.empty()) {
                EquivalenceClass current = resultParts.back();
                resultParts.pop_back();
                bool merged = false;
                for (auto& part : newResultParts) {
                    if (current.lowerBound[field] <= part.upperBound[field]
&&
                        current.upperBound[field] >=
part.lowerBound[field]) {
                        part.lowerBound[field] =
std::min(current.lowerBound[field], part.lowerBound[field]);
                        part.upperBound[field] =

```



```

std::max(current.upperBound[field], part.upperBound[field]);
        merged = true;
        break;
    }
}
if (!merged) {
    newResultParts.push_back(current);
}
}
resultParts = newResultParts;
}

EquivalenceClass mergedResult = resultParts[0];
for (size_t j = 1; j < resultParts.size(); ++j) {
    for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
        mergedResult.lowerBound[i]
std::min(mergedResult.lowerBound[i], resultParts[j].lowerBound[i]);
        mergedResult.upperBound[i]
std::max(mergedResult.upperBound[i], resultParts[j].upperBound[i]);
    }
}
for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++) {
    if (mergedResult.lowerBound[i] > mergedResult.upperBound[i])
{
        uint64_t temp = mergedResult.lowerBound[i];
        mergedResult.lowerBound[i] = mergedResult.upperBound[i];
        mergedResult.upperBound[i] = temp;
    }
}

return mergedResult;
} else {
    return EquivalenceClass();
}
}

bool EquivalenceClass::isEmpty() const {
    for (int i = 0; i < ALL_FIELD_INDEX_END_MARKER; ++i) {
        if (this->lowerBound[i] != 0 || this->upperBound[i] != 0) {
            return false;
        }
    }
}

```

as_switch.py

```

from ryu.base import app_manager

```

```

from ryu.base.app_manager import lookup_service_brick
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, CONFIG_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet, arp, ipv4
import utils.flowmod
import utils.ipv4
from utils.flowmod import send_flow_mod
from network_awareness import NetworkAwareness
import json
import os
import math

class RoutingSwitch(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]

    _CONTEXTS = {
        'network_awareness': NetworkAwareness
    }

    def __init__(self, *args, **kwargs):
        super(RoutingSwitch, self).__init__(*args, **kwargs)
        self.network_awareness = kwargs['network_awareness']
        self.dpid_mac_port = {}
        with open(os.environ["CONFIG"], "r") as f:
            self.routing_cfg = json.load(f)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def packet_in_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        parser = dp.ofproto_parser

        dpid = dp.id
        in_port = msg.in_port

        pkt = packet.Packet(msg.data)
        eth_pkt = pkt.get_protocol(ethernet.ethernet)
        arp_pkt = pkt.get_protocol(arp.arp)
        ipv4_pkt = pkt.get_protocol(ipv4.ipv4)

```

```

pkt_type = eth_pkt.ethertype

# layer 2 self-learning
dst_mac = eth_pkt.dst
src_mac = eth_pkt.src

self.dpid_mac_port.setdefault(dpid, {})
self.dpid_mac_port[dpid][src_mac] = in_port

if isinstance(arp_pkt, arp.arp):
    self.handle_arp(msg, in_port, dst_mac, pkt_type)

if isinstance(ipv4_pkt, ipv4.ipv4):
    self.handle_ipv4(msg, dpid, in_port, src_mac, dst_mac,
ipv4_pkt.src, ipv4_pkt.dst, pkt_type)

def handle_arp(self, msg, in_port, dst_mac, pkt_type):
    dp = msg.datapath
    ofp = dp.ofproto
    parser = dp.ofproto_parser

    dpid = dp.id

    if dst_mac in self.dpid_mac_port[dpid]:
        out_port = self.dpid_mac_port[dpid][dst_mac]
        actions = [parser.OFPActionOutput(out_port)]
        out = parser.OFPPacketOut(
            datapath=dp, buffer_id=msg.buffer_id, in_port=in_port,
actions=actions, data=msg.data)
        dp.send_msg(out)
    else:
        # send to the switch port which linked hosts
        for d, ports in self.network_awareness.port_info.items():
            for p in ports:
                # except the source host
                if d == dpid and p == in_port:
                    continue

                dp = self.network_awareness.switch_info[d]
                actions = [parser.OFPActionOutput(p)]
                out = parser.OFPPacketOut(
                    datapath=dp, buffer_id=msg.buffer_id,
in_port=ofp.OFPP_CONTROLLER, actions=actions, data=msg.data)
                dp.send_msg(out)

```



```

def handle_ipv4(self, msg, dpid, in_port, src_mac, dst_mac, src_ip,
dst_ip, pkt_type):
    print(f"Packet to {dpid}")
    parser = msg.datapath.ofproto_parser

    switch_net = self.routing_cfg["switch_nets"][dpid]
    srcnet = None
    dstnet = None

    for net in self.routing_cfg["gateways"]:
        if utils.ipv4.in_net(net, src_ip):
            srcnet = net
            break

    if utils.ipv4.in_net(switch_net, dst_ip):
        # Local switching. Install shortest paths.
        dstnet = switch_net
        gateways = None
    else:
        # Routing. Go to the closest gateway
        for dst_candidate in
self.routing_cfg["gateways"][switch_net]:
            if utils.ipv4.in_net(dst_candidate, dst_ip):
                dstnet = dst_candidate
                gateways =
self.routing_cfg["gateways"][switch_net][dstnet]

    if not srcnet or not dstnet:
        print("src / dst not recognized, unable to forward.")
        return

    # add_path should be called only from switch to switch / host.
    def add_path(route, dl_src, dl_dst, nw_src, nw_dst, priority=5):
        port_path = []
        for i in range(len(route) - 1):
            out_port = self.network_awareness.link_info[(route[i],
route[i + 1])]
            port_path.append((route[i], out_port))
        self.show_path(route[0], route[-1], port_path)

    for node in port_path:
        waypoint_dpid, out_port = node

        send_flow_mod(waypoint_dpid, dl_src, dl_dst, nw_src,

```

```

nw_dst, None, out_port, priority)

        src_net_16 = self.get_subnet_16(nw_src)
        dst_net_16 = self.get_subnet_16(nw_dst)
        high_priority = 10
        send_flow_mod(waypoint_dpid, dl_src, dl_dst, src_net_16,
dst_net_16, None, out_port, high_priority)

        # return the output port of current switch.
        return port_path[0][1]

    if gateways is None:
        # Handle packet switching within AS.
        dpid_path = self.network_awareness.shortest_path(dpid,
dst_ip)

        if not dpid_path:
            return

        # get port path: h1 -> in_port, s1, out_port -> h2
        out_port = add_path(dpid_path, None, None, srcnet, dstnet)
    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]
            out_port = add_path(route, None, None, srcnet, dstnet)
        else:
            # otherwise, send to the closest gateway
            min_delay = math.inf
            min_gw = None
            for gw in gateways:
                delay = self.network_awareness.shortest_path_length(dpid, gw)
                if delay < 0:
                    continue

                if delay < min_delay:
                    min_delay = delay
                    min_gw = gw

            if min_gw is None:
                # no way to gateway
                return

```

```

        dpid_path = self.network_awareness.shortest_path(dpid,
min_gw)

        if not dpid_path:
            return

        out_port = add_path(dpid_path, None, None, srcnet, dstnet)

        # after installing the path, send the packet
        dp = self.network_awareness.switch_info[dpid]
        actions = [parser.OFPActionOutput(out_port)]
        out = parser.OFPPacketOut(
            datapath=dp,    buffer_id=msg.buffer_id,    in_port=in_port,
actions=actions, data=msg.data)
        dp.send_msg(out)

def get_subnet_16(self, ip):
    parts = ip.split('.')
    return f"{parts[0]}.{parts[1]}.0.0/16"

def show_path(self, src, dst, port_path):
    self.logger.info('path: {} -> {}'.format(src, dst))
    path = str(src) + ' -> '
    for node in port_path:
        path += 's{}:{}'.format(*node) + ' -> '
    path += str(dst)
    self.logger.info(path)
    self.logger.info('\n')

```