

Software Define Network Lab4

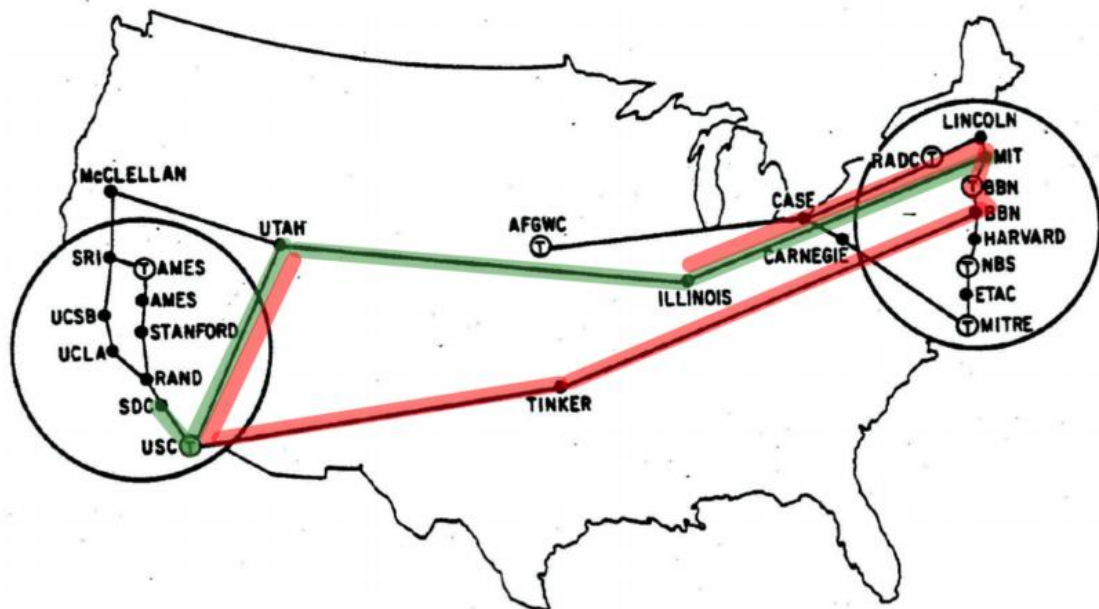
实验目的:

理解网络故障的普遍性
熟悉网络验证工具 VeriFlow 的原理
掌握 VeriFlow 的故障检测方法

实验环境:

Windows 10, VMware Workstation Pro, Ubuntu

问题背景:



TINKER 处建立了一个流量分析中心，为了保证 UTAH 和 ILLINOIS 之间的流量可以在 TINKER 进行分析，你下发了图中的红色路径。你的同事 Bob 接到了另外一个需求，要求建立从 SDC 到 MIT 跳数最少的路径，即图中的绿色路径。不同的路径需求来自不同的用户，没有经过协调，产生了一个转发环路。请你运行 VeriFlow 工具，对上述两条转发路径进行检查。

实验过程:

1. 示例：如何生成转发环路？

在命令行终端内输入 `sudo python Arpanet19723.py`，启动网络拓扑。

```
sdn@ubuntu: ~/Desktop/mininet/custom
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python Arpanet19723.py
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Adding hosts:
AFGWC AMES AMES13 BBN BBN15 CARNEGIE CASE ETAC HARVARD ILLINOIS L
incoln MIT MITRE McClellan NBS RADC RAND SDC SRI Stanford Tinker
UCLA UCSB USC UTAH
*** Adding switches:
s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 s16 s17 s18 s1
9 s20 s21 s22 s23 s24 s25
*** Adding links:
```

在命令行终端内输入 `ryu-manager ofctl_rest.py shortest_path.py --observe-links`，启动最短路径的控制程序。

```
sdn@ubuntu: ~/Desktop/mininet/custom
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager ofctl_rest.py shor
test_path.py --observe-links
loading app ofctl_rest.py
loading app shortest_path.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
```

在 mininet 命令行内输入 `SDC ping MIT -c5`，建立 SDC 与 MIT 之间的链接。

```
*** Starting CLI:
mininet> SDC ping MIT -c5
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=184 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=234 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=232 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=234 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 4 received, 20% packet loss, time 4034ms
rtt min/avg/max/mdev = 183.597/220.828/234.317/21.512 ms
mininet> SDC ping MIT -c5
```

可以看到，此时 SDC ping MIT 能 ping 通，两主机之间的连通性没有问题。

```
(237799) wsgi starting up on http://0.0.0.0:8080
path: 10.0.0.18 -> 10.0.0.12
10.0.0.18 -> 1:s15:3 -> 3:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3
:s25:1 -> 10.0.0.12

(237799) accepted ('127.0.0.1', 53554)
127.0.0.1 - - [04/May/2023 09:09:25] "POST /stats/flowentry/add HTTP/1.1" 200 139 0.006510
<Response [200]>
```

Ryu 控制器内也可以看到打印的路径信息与网络信息。

在命令行终端内输入 `sudo python waypoint_path.py`, 下发从 UTAH 途经 TINKER 到达 ILLINOIS 的路径。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python waypoint_path.py
[sudo] password for sdn:
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
install waypoint path: 23 -> 1
23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:2 -> 1
```

随后在 mininet 终端内输入 `SDC ping MIT -c5`, 测试此时两主机之间的连通性。

```
mininet> SDC ping MIT -c5
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 0 received, 100% packet loss, time 4073ms

mininet> █
```

可以看到, 此时两主机之间无法 ping 通。

在命令行终端内输入 `sudo ovs-ofctl dump-flows s22`, 查看 s22 交换机的流表。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo ovs-ofctl dump-flows s22
cookie=0x0, duration=430.370s, table=0, n_packets=318, n_bytes=19080, priority=655
35,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=425.373s, table=0, n_packets=19, n_bytes=1862, priority=1,ip,
in_port="s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth3"
cookie=0x0, duration=199.447s, table=0, n_packets=596, n_bytes=58408, priority=10,
ip,in_port="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-eth
2"
cookie=0x0, duration=199.441s, table=0, n_packets=2328, n_bytes=228144, priority=1
0,ip,in_port="s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s22-e
th4"
```

可以看到, 匹配某一条流表项的数据包数目异常大, 此时拓扑中存在环路。

2. 使用 VeriFlow

在命令行终端内输入 `git clone https://github.com/samueljero/BEADS.git`

```
98324@MSIGS66 MINGW64 ~/Desktop/Text/Software Define Network/Lab4
$ git clone https://github.com/samueljero/BEADS.git
Cloning into 'BEADS'...
remote: Enumerating objects: 1884, done.
remote: Total 1884 (delta 0), reused 0 (delta 0), pack-reused 1884
Receiving objects: 100% (1884/1884), 4.15 MiB | 364.00 KiB/s, done.
Resolving deltas: 100% (1027/1027), done.
```

在命令行终端内输入 `cd BEADS`, 切换到/BEADS 目录下, 然后将补丁文件保

存到当前目录下，输入 `git am 0001-for-xjtu-sdn-exp-2020.patch`，打上补丁文件。

```
98324@MSIGS66 MINGW64 ~/Desktop/Text/Software Define Network/Lab4/BEADS (master)
$ git am 0001-for-xjtu-sdn-exp-2020.patch
.git/rebase-apply/patch:158: trailing whitespace.
.git/rebase-apply/patch:217: trailing whitespace.
warning: 2 lines add whitespace errors.
Applying: for xjtu-sdn-exp-2020
```

在 BEADS/veriflow/VeriFlow 目录下，打开命令行终端，输入 `make clean all`，编译 VeriFlow。

```
sdn@ubuntu:~/Desktop/BEADS/veriflow/VeriFlow$ make clean all
rm -f VeriFlow.o OpenFlowProtocolMessage.o Network.o EquivalenceClass.o Forwardi
ngGraph.o ForwardingLink.o TrieNode.o Rule.o Trie.o EquivalenceRange.o Test.o ne
t.o thread.o StringTokenizer.o VeriFlow *.o
g++ -O2 -g -Wall -fmessage-length=0 -D_REENTRANT -std=c++11 -c -o VeriFlow.o V
eriFlow.cpp
VeriFlow.cpp: In member function 'bool VeriFlow::verifyRule(const Rule&, int, do
uble&, double&, double&, double&, long unsigned int&, FILE*)':
```

在命令行终端内输入 `ryu-manager ofctl_rest.py shortest_path.py --ofp-tcp-listen-port 1024 --observe-links`，在自定义端口开启远程控制器，运行最短路程序。

```
sdn@ubuntu:~/Desktop/mininet/custom$ ryu-manager ofctl_rest.py shor
test_path.py --ofp-tcp-listen-port 1024 --observe-links
loading app ofctl_rest.py
loading app shortest_path.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.controller.ofp_handler
instantiating app None of DPSet
```

在命令行终端内输入 `./VeriFlow 6633 127.0.0.1 1024 Arpanet19723.txt log_file.txt`，启动 VeriFlow 的 Proxy 模式。

```
sdn@ubuntu:~/Desktop/BEADS/veriflow/VeriFlow$ ./VeriFlow 6633 127.0.0.1 1024
Arpanet19723.txt log_file.txt
id 125 ipAddress 10.0.0.25 endDevice 1 port 0 nextHopIpAddress 20.0.0.23
id 123 ipAddress 10.0.0.23 endDevice 1 port 0 nextHopIpAddress 20.0.0.19
id 124 ipAddress 10.0.0.24 endDevice 1 port 0 nextHopIpAddress 20.0.0.22
id 117 ipAddress 10.0.0.17 endDevice 1 port 0 nextHopIpAddress 20.0.0.13
```

在命令行终端内输入 `sudo python Arpanet19723.py`，启动拓扑。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python Arpanet19723.py
*** Creating network
*** Adding controller
*** Adding hosts:
AFGWC AMES AMES13 BBN BBN15 CARNEGIE CASE ETAC HARVARD ILLINOIS L
incoln MIT MITRE McClellan NBS RADC RAND SDC SRI Stanford Tinker
UCLA UCSB USC UTAH
*** Adding switches:
```

在 mininet 命令行内输入 `SDC ping MIT -c5`，建立 SDC 与 MIT 之间的链接。

```
mininet> SDC ping MIT -c5
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
64 bytes from 10.0.0.12: icmp_seq=1 ttl=64 time=231 ms
64 bytes from 10.0.0.12: icmp_seq=2 ttl=64 time=231 ms
64 bytes from 10.0.0.12: icmp_seq=3 ttl=64 time=231 ms
64 bytes from 10.0.0.12: icmp_seq=4 ttl=64 time=231 ms
64 bytes from 10.0.0.12: icmp_seq=5 ttl=64 time=232 ms

--- 10.0.0.12 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4005ms
rtt min/avg/max/mdev = 230.800/231.255/231.597/0.285 ms
```

可以看到，此时 SDC ping MIT 能 ping 通，两主机之间的连通性没有问题。

```
log_file.txt
1
2 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class as current location (20.0.0.001.1) not found in the graph.
3 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_l_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_l_dst (165252221582-01:80:c2:00:00:0e, 165252221582-01:80:c2:00:00:0e), nw_src (0-0.0.0.0, 4294967295-255.255.255.255), nw_dst (0-0.0.0.0, 4294967295-255.255.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (165252221582, 165252221582), Field 3 (35020, 35020), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (0, 4294967295), Field 9 (0, 4294967295), 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 the following packet class as current location (20.0.0.001.1) not found in the graph.
7 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] d_l_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff), d_l_dst (0-00:00:00:00:00:00, 165252221581-01:80:c2:00:00:0d), nw_src (0-0.0.0.0, 4294967295-255.255.255.255), nw_dst (0-0.0.0.0, 4294967295-255.255.255.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (0, 165252221581), Field 3 (0, 65535), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (0, 4294967295), Field 9 (0, 4294967295), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
8
```

查看 log_file.txt 可以看到，此时 VeriFlow 只检测到了黑洞(Black Hole)，没有检测到环路(Loop)。

在命令行终端内输入 `sudo python waypoint_path.py`，下发从 UTAH 途经 TINKER 到达 ILLINOIS 的路径。

```
sdn@ubuntu:~/Desktop/mininet/custom$ sudo python waypoint_path.py
[sudo] password for sdn:
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
install waypoint path: 23 -> 1
23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:2 -> 1
```

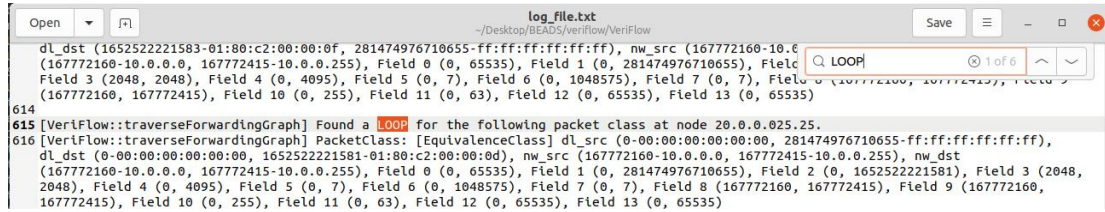
再次在 mininet 命令行内输入 SDC ping MIT，测试两主机连通性。

```
mininet> SDC ping MIT
PING 10.0.0.12 (10.0.0.12) 56(84) bytes of data.
^C
--- 10.0.0.12 ping statistics ---
634 packets transmitted, 0 received, 100% packet loss, time 648193ms

mininet>
```

可以看到，此时两主机之间无法 ping 通，并且在 log 文件中观察 VeriFlow 检测到的环路信息。

打开 BEADS/veriflow/VeriFlow 目录下的 log_file.txt 文件，可以看到 VeriFlow 检测到的环路信息。



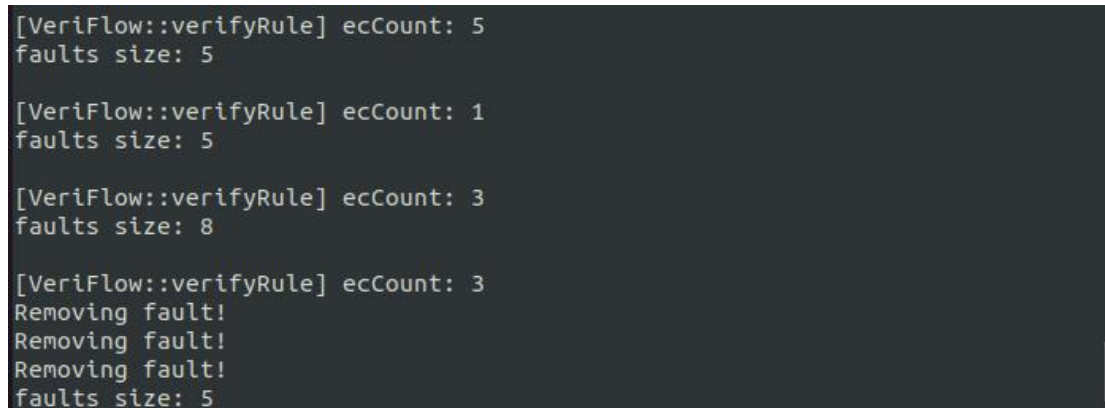
3. 基础实验：EC 数目的打印

对每条验证的规则，实验要求输出这条规则所影响的 EC 数目。

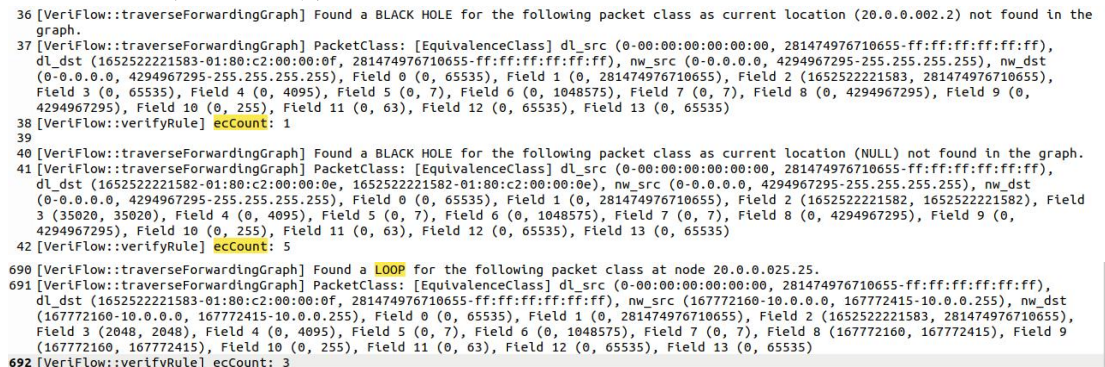
查看源码可以知道，打印 EC 数目的函数在 VeriFlow.cpp 中，为 VeriFlow::verifyRule() 函数。模仿 ecCount == 0 的代码格式，修改此部分代码如下。

```
1017     else
1018     {
1019         fprintf(stdout, "\n");
1020         fprintf(stdout, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount); // output to the veriflow console
1021         fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount); // output to the FILE *fp, which means the log_file.txt
1022     }
```

修改完成后重新进行 VeriFlow 的编译、启动 Ryu 控制器、启动 VeriFlow 的 Proxy 模式、启动拓扑、建立 SDC 与 MIT 之间的链接、下发路径、再次测试两主机连通性、查看 log_file.txt。



可以看到，VeriFlow 命令行终端打印了受影响的 EC 数目，一开始在 1 和 5 之间变化，稳定后保持为 3。



在 log_file.txt 中也可以看到相同记录。

4. 基础实验：环路路径的打印

要求打印出环路的信息，包括出现环路的提示信息，EC 的基本信息和环路路径上的 IP 地址。提示: traverseForwardingGraph 函数中的 visited 为 unordered_set,

可改成有序的数据结构。

按照实验指导书的提示，`traverseForwardingGraph` 函数的作用是遍历某个特定 EC 的转发图，验证是否存在环路或黑洞。为了记录环路路径，我们建立一个字符向量 `vector<string> path` 用来保存每次的 `currentLocation`。

`traverseForwardingGraph` 函数的第一次调用是在 `VeriFlow::verifyRule` 函数中，我们建立一个字符向量并修改 `traverseForwardingGraph` 函数，将字符向量 `path` 作为参数传入函数中。

```
1047     for(unsigned int i = 0; i < vGraph.size(); i++)
1048     {
1049         unordered_set< string > visited;
1050         vector<string> path; // add a string vector as a parameter
1051         string lastHop = network.getNextHopIpAddress(rule.location,rule.in_port);
1052         // fprintf(fp, "start traversing at: %s\n", rule.location.c_str());
1053         if(!this->traverseForwardingGraph(vFinalPacketClasses[i], vGraph[i], rule.location, lastHop, visited, path, fp)) {
1054             // pass the parameter to the function
1055             ++currentFailures;
1056         }
1057     }
```

因为我们修改了 `traverseForwardingGraph` 函数的定义，所以在 `VeriFlow.h` 文件中，也要修改函数对应的声明。

```
111     bool traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, const string&
112     lastHop, unordered_set< string > visited, vector<string>& path, FILE* fp);
```

`traverseForwardingGraph` 函数定义在 `VeriFlow.cpp` 文件中，我们要修改其函数的定义。

```
1113 // modify the arguments
1114 bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, const
1115 string& lastHop, unordered_set< string > visited, vector<string>& path, FILE* fp)
1116 {
```

为了打印环路路径，我们定义了一个 `printLoop` 函数，用于打印环路路径 `path` 上结点的信息与当前结点 `current` 的信息。

```
1108 // print loop and information
1109 static void printLoop(FILE* fp, const vector<string>& path, const string& current)
1110 {
1111     fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");
1112     int n = path.size(); // get the number of nodes on loop
1113     for(int i = 0; i < n; i++)
1114     {
1115         fprintf(fp, "%s -> ", path[i].c_str()); // print path nodes
1116     }
1117     fprintf(fp, "%s\n", current.c_str()); // print current node
1118 }
```

在 `traverseForwardingGraph` 函数中，如果存在环路，我们就调用定义的 `printLoop` 函数。

```
1139     if(visited.find(currentLocation) != visited.end())
1140     {
1141         // Found a loop.
1142         fprintf(fp, "\n");
1143         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the
1144         following packet class at node %s.\n", currentLocation.c_str());
1145         fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n",
1146         packetClass.toString().c_str());
1147         //printloop
1148         fprintf(fp, "curr:%s\n", currentLocation.c_str());
1149         printLoop(fp, path, currentLocation);
1150     }
```

为了判断当前结点 `currentLocation` 是否在环路路径 `path` 中，我们定义了一个 `isInPath` 函数，用于遍历环路路径 `path`，确定当前结点是否在环路路径中。

```

1094 // check if currentLocation is in path
1095 static bool isInPath(const vector<string>& path, const string& location)
1096 {
1097     int n = path.size(); //// get the number of nodes on loop
1098     for(int i = 0 ;i < n;i++) // go through the path
1099     {
1100         if(path[i] == location)
1101         {
1102             return true;
1103         }
1104     }
1105     return false;
1106 }

```

在 `traverseForwardingGraph` 函数中,如果存在环路且当前结点 `currentLocation` 不在环路路径 `path` 中,就将其添加到 `path` 中。

```

1159     visited.insert(currentLocation);
1160     if(!isInPath(path, currentLocation))
1161     {
1162         path.push_back(currentLocation); // If not, add currentLocation to path
1163     }

```

修改完成后重新进行 VeriFlow 的编译、启动 Ryu 控制器、启动 VeriFlow 的 Proxy 模式、启动拓扑、建立 SDC 与 MIT 之间的链接、下发路径、再次测试两主机连通性、查看 `log_file.txt`。

```

694 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.025.25.
695 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src (0-00:00:00:00:00:00, 281474976710655-ff:ff:ff:ff:ff:ff),
dl_dst (165252221583-01:80:c2:00:00:0f, 281474976710655-ff:ff:ff:ff:ff:ff), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst
(167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (0, 281474976710655), Field 2 (165252221583, 281474976710655),
Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575), Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9
(167772160, 167772415), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
696 [VeriFlow::traverseForwardingGraph] Loop path is:
697 20.0.0.025.25 -> 20.0.0.001.1 -> 20.0.0.023.23 -> 20.0.0.022.22 -> 20.0.0.009.9 -> 20.0.0.016.16 -> 20.0.0.007.7 -> 20.0.0.025.25
698 [VeriFlow::verifyRule] ecCount: 3
699

```

可以看到, `log_file.txt` 中打印了环路的信息,包括出现环路的提示信息,EC 的基本信息和环路路径上的 IP 地址。

5. 基础实验: 相关数据包信息的打印

EC 的基本信息显示为 14 个域的区间形式,为方便 Bob 查错,现简化 EC 信息的表示形式,仅从 14 个域中提取 TCP/IP 五元组作为主要信息显示。提示:在环路路径打印的基础上,修改 EC 的显示格式。

```

1140     if(visited.find(currentLocation) != visited.end())
1141     {
1142         // Found a loop.
1143         fprintf(fp, "\n");
1144         fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a loop for the following packet class at node %s.\n", currentLocation.c_str());
1145         fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());

```

查看源码可以发现,打印 EC 信息的代码在 `VeriFlow.cpp` 文件的 `traverseForwardingGraph` 函数中,将传入的参数 `packetClass` 使用 `toString` 方法转换为字符串进行输出。

```

111     bool traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, const string&
lastHop, unordered_set< string> visited, vector<string>& path, FILE* fp);
112

```

查看 `traverseForwardingGraph` 函数的定义我们可以知道, `packetClass` 参数是 `EquivalenceClass` 类型,我们需要修改 `EquivalenceClass` 类内定义的 `toString` 方法。

`EquivalenceClass` 类定义在 `EquivalenceClass.cpp` 文件中,按照实验指导书内的样例格式,我们修改类中的 `toString` 方法,修改后代码如下。


```

110 // Modify the toString method of EquivalenceClass
111 string EquivalenceClass::toString() const
112 {
113     char buffer[1024];
114     // sprintf(buffer, "[EquivalenceClass] dl_src (%lu-%s, %lu-%s), dl_dst (%lu-%s, %lu-%s)",
115     // this->lowerBound[DL_SRC], ::getMacValueAsString(this->lowerBound[DL_SRC]).c_str(),
116     // this->upperBound[DL_SRC], ::getMacValueAsString(this->upperBound[DL_SRC]).c_str(),
117     // this->lowerBound[DL_DST], ::getMacValueAsString(this->lowerBound[DL_DST]).c_str(),
118     // this->upperBound[DL_DST], ::getMacValueAsString(this->upperBound[DL_DST]).c_str());
119     sprintf(buffer, "nw_src(%s-%s), nw_dst(%s-%s), nw_proto(%lu-%lu), nw_proto(%lu-%lu), tp_src(%lu-%lu), tp_dst(%lu-%lu)",
120     ::getIpValueAsString(this->lowerBound[NW_SRC]).c_str(),
121     ::getIpValueAsString(this->upperBound[NW_SRC]).c_str(),
122     ::getIpValueAsString(this->lowerBound[NW_DST]).c_str(),
123     ::getIpValueAsString(this->upperBound[NW_DST]).c_str(),
124     this->lowerBound[NW_PROTO],
125     this->upperBound[NW_PROTO],
126     this->lowerBound[TP_SRC],
127     this->upperBound[TP_SRC],
128     this->lowerBound[TP_DST],
129     this->upperBound[TP_DST]);
130     string retVal = buffer;
131     return retVal;
132 }

```

修改完成后重新进行 VeriFlow 的编译、启动 Ryu 控制器、启动 VeriFlow 的 Proxy 模式、启动拓扑、建立 SDC 与 MIT 之间的链接、下发路径、再次测试两主机连通性、查看 log_file.txt。

```

694 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.025.25.
695 [VeriFlow::traverseForwardingGraph] PacketClass: nw_src(10.0.0.0-10.0.0.255), nw_dst(10.0.0.0-10.0.0.255), nw_proto(0-255),
nw_proto(0-65535), tp_src(0-65535), tp_dst(140300928295696-0)
696 [VeriFlow::traverseForwardingGraph] Loop path is:
697 20.0.0.025.25 -> 20.0.0.001.1 -> 20.0.0.022.22 -> 20.0.0.009.9 -> 20.0.0.016.16 -> 20.0.0.007.7 -> 20.0.0.025.25
698 [VeriFlow::verifyRule] ecCount: 3

```

可以看到，log_file.txt 内打印了简化后的 EC 信息。

6. 分析原始代码与补丁代码的区别，思考为何需要添加补丁

在/BEADS 目录下打开命令行，输入 git diff HEADS origin/HEADS，查看补丁修改的文件内容。

```

Windows PowerShell
PS C:\Users\98324\Desktop\Text\Software Define Network\Lab4\test\BEADS> git diff HEADS origin/HEADS
diff --git a/veriflow/VeriFlow/Network.cpp b/veriflow/VeriFlow/Network.cpp
index befaade..e3d8d15 100644
--- a/veriflow/VeriFlow/Network.cpp
+++ b/veriflow/VeriFlow/Network.cpp
@@ -18,7 +18,6 @@
#include <cstdio>
#include <string>
#include <unordered_map>
-#include <sstream>
#include "Network.h"
#include "ForwardingDevice.h"

diff --git a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
index ac5084d..a081d3b 100644
--- a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
+++ b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
@@ -292,10 +292,8 @@ void OpenFlowProtocolMessage::processFlowRemoved(const char* data, ProxyConnecti
    rule.type = FORWARDING;
    rule.wildcards = ntohl(ofr->match.wildcards);
-    rule.fieldValue[IN_PORT] = "0"; //::convertIntToString(ntohs(ofr->match.in_port));

```

红色的内容是打上补丁后增加的内容，即原始文件相比于打上补丁的文件后减少的内容。其中，比较重要的修改内容如下。

```
Windows PowerShell
#include "ForwardingDevice.h"

diff --git a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
index ac5084d..a081d3b 100644
--- a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
+++ b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
@@ -292,10 +292,8 @@ void OpenFlowProtocolMessage::processFlowRemoved(const char* data, ProxyConnecti
    rule.type = FORWARDING;
    rule.wildcards = ntohl(ofr->match.wildcards);

-    rule.fieldValue[IN_PORT] = "0"; // ::convertIntToString(ntohs(ofr->match.in_port));
-    rule.fieldMask[IN_PORT] = "0"; // ((rule.wildcards == OFPPFW_ALL) || ((rule.wildcards & OFPPFW_IN_PORT) != 0)) ? "0" : "65535";
+    rule.in_port = ntohs(ofr->match.in_port);
+    rule.fieldValue[IN_PORT] = ::convertIntToString(ntohs(ofr->match.in_port));
+    rule.fieldMask[IN_PORT] = ((rule.wildcards == OFPPFW_ALL) || ((rule.wildcards & OFPPFW_IN_PORT) != 0)) ? "0" : "65535";

    rule.fieldValue[DL_SRC] = ::getMacValueAsString(ofr->match.dl_src);
    rule.fieldMask[DL_SRC] = ((rule.wildcards == OFPPFW_ALL) || ((rule.wildcards & OFPPFW_DL_SRC) != 0)) ? "0:0:0:0:0:0" : "FF:FF:FF:FF:FF:FF";
@@ -348,13 +346,12 @@ void OpenFlowProtocolMessage::processFlowRemoved(const char* data, P
```

在 OpenFlowProtocolMessage.cpp 中增加了对 rule 对象中 in_port 成员的处理。

```
Windows PowerShell
diff --git a/veriflow/VeriFlow/Rule.cpp b/veriflow/VeriFlow/Rule.cpp
index 847d902..a0ec591 100644
--- a/veriflow/VeriFlow/Rule.cpp
+++ b/veriflow/VeriFlow/Rule.cpp
@@ -36,7 +36,6 @@ Rule::Rule()
    this->location = "";
    this->nextHop = "";
-    this->in_port = 65536;
    this->priority = INVALID_PRIORITY;
    // this->outPort = OFPP_NONE;
}
@@ -55,7 +54,6 @@ Rule::Rule(const Rule& other)
    this->location = other.location;
    this->nextHop = other.nextHop;
-    this->in_port = other.in_port;
    this->priority = other.priority;
    // this->outPort = other.outPort;
}
@@ -181,7 +179,6 @@ bool Rule::equals(const Rule& other) const
    if((this->type == other.type)
        && (this->wildcards == other.wildcards)
        && (this->location.compare(other.location) == 0)
-        && (this->in_port == other.in_port))
```

在 Rule.cpp 的 Rule 类中增加了 in_port 成员，在多种构造函数中增加了对 in_port 对象的初始化和存储。而原始文件没有区分 in_port，这样如果两个数据包的转发路径存在重叠的部分，会被认为是相同的数据通路，显然这样的方式划分的等价类和我们的目标是不一致的，会将等价类划分地更大，因此需要添加 in_port 来保证正确地划分等价类。

```

vGraph.push_back(graph);
}
@@ -1045,9 +1043,7 @@ bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime, double&
for(unsigned int i = 0; i < vGraph.size(); i++)
{
    unordered_set< string > visited;
    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, visited, fp)) {
+    if(!this->traverseForwardingGraph(vFinalPacketClasses[i], vGraph[i], rule.location, visited, fp)) {
        ++currentFailures;
    }
}
@@ -1088,10 +1084,8 @@ bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime, double&
return true;
}

+bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, const string& lastHop, unordered_set< string > visited, FILE* fp)
+bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, unordered_set< string > visited, FILE* fp)

```

在 VeriFlow.cpp 文件的 verifyRule 函数中增加了对 lastHop 的存储，提高了查找黑洞的能力。

```

+bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, const string& lastHop, unordered_set< string > visited, FILE* fp)
+bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, unordered_set< string > visited, FILE* fp)
{
    // fprintf(fp, "traversing at node: %s\n", currentLocation.c_str());
    if(graph == NULL)
    {
        /* fprintf(fp, "\n");
@@ -1119,7 +1113,7 @@ bool VeriFlow::traverseForwardingGraph(const EquivalenceClass& packetClass, ForwardingGraph* graph, const string& currentLocation, unordered_set< string > visited, FILE* fp)
    }
    faults.push_back(packetClass);
    return false;
}

```

在 VeriFlow.cpp 文件中修改了 traverseForwardingGraph 函数的定义，增加了 lastHop 参数，提高了查找黑洞的能力。


```
Windows PowerShell

list< ForwardingLink >::const_iterator itr = linkList.begin();
// input_port as a filter
if(lastHop.compare("NULL") == 0 || itr->rule.in_port == 65536){
    // 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 class as there is no outgoing link at current location (%s).\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);
    }
}
```

在 VeriFlow.cpp 文件的 traverseForwardingGraph 函数中增加了对黑洞的一种判断，完善了对黑洞的判断情况。

在 github 中开源代码中，给出了两种判断黑洞的情况：

1. 当前交换机或主机并不在当前网络中
2. 当前交换机或主机在网络中，但是无链路与其他交换机或主机相连

补丁中增加了一种判断黑洞的情况：

当前的交换机或者主机在网络的拓扑结构中，也存在与它相连的链路，但由于网络结构变化，使得从当前的交换机或者主机所在位置和相应端口(in_port)，找不到上一跳的交换机或者主机。

如果 lastHop 不是 NULL 且 in_port 不是 65536，则遍历当前结点的链接列表，根据当前交换机或主机的结点位置 and 对应端口查找上一跳(lastHop)的结点，如果遍历完成后仍找不到与 lastHop 对应的主机，则认为存在黑洞。

此外还有一些 VeriFlow.h 中关于 traverseForwardingGraph 函数声明的修改与调用 traverseForwardingGraph 函数时传入参数的修改。

准备知识：环路产生原因分析

```
path: 10.0.0.18 -> 10.0.0.12
10.0.0.18 -> 1:s15:3 -> 3:s22:4 -> 4:s23:2 -> 3:s1:2 -> 2:s25:1 -> 10.0.0.12
```

在 mininet 命令行内输入 SDC ping MIT 后，可以在 Ryu 控制器里看到，我们建立了一条这样的路径，从 10.0.0.18 到 10.0.0.12。

```

sdn@ubuntu:~/Desktop/mininet/custom$ sudo python waypoint_path.py
[sudo] password for sdn:
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
install waypoint path: 23 -> 1
23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:2 -> 1

```

而下发路径后，我们可以看到，新增路径从交换机 s23 到交换机 s1。新建立的链接路径也经过了交换机 s22。交换机 s23 连接到交换机 s22 的端口 4，再由交换机 s22 的端口 2 连接到交换机 s9。

两条路径的重合结点为 s22，s23，s1，s25。

查看网络拓扑图可以看到，两条路径的重合结点是 MIT 主机所连接的交换机与 USC 主机所连接的交换机。

在 mininet 命令行内输入 dump，查看各结点详细网络信息。

```

<CPULimitedHost ILLINOIS: ILLINOIS-eth0:10.0.0.10 pid=71833>
<CPULimitedHost Lincoln: Lincoln-eth0:10.0.0.11 pid=71837>
<CPULimitedHost MIT: MIT-eth0:10.0.0.12 pid=71841>
<CPULimitedHost MITRE: MITRE-eth0:10.0.0.13 pid=71845>
<CPULimitedHost McClellan: McClellan-eth0:10.0.0.14 pid=71849>
<CPULimitedHost NBS: NBS-eth0:10.0.0.15 pid=71853>
<CPULimitedHost RADC: RADC-eth0:10.0.0.16 pid=71857>
<CPULimitedHost RAND: RAND-eth0:10.0.0.17 pid=71861>
<CPULimitedHost SDC: SDC-eth0:10.0.0.18 pid=71865>
<CPULimitedHost SRI: SRI-eth0:10.0.0.19 pid=71869>
<CPULimitedHost Stanford: Stanford-eth0:10.0.0.20 pid=71873>
<CPULimitedHost Tinker: Tinker-eth0:10.0.0.21 pid=71877>
<CPULimitedHost UCLA: UCLA-eth0:10.0.0.22 pid=71881>
<CPULimitedHost UCSB: UCSB-eth0:10.0.0.23 pid=71885>
<CPULimitedHost USC: USC-eth0:10.0.0.24 pid=71889>
<CPULimitedHost UTAH: UTAH-eth0:10.0.0.25 pid=71893>

```

可以看到，MIT 主机的 IP 地址为 10.0.0.12，SDC 主机的 IP 地址为 10.0.0.18，USC 主机的 IP 地址为 10.0.0.24。MIT 主机连接到交换机 s15，交换机 s15 通过端口 3 连接到交换机 s22 的端口 3，再由交换机 s22 的端口 4 连接到交换机 s23 的 4 端口。

更具体地，我们在 mininet 命令行内输入 links，查看各结点之间连接的信息。

```

s22-eth1<->USC-eth0 (OK OK)
s22-eth4<->s23-eth4 (OK OK)
s23-eth1<->UTAH-eth0 (OK OK)
s24-eth1<->Lincoln-eth0 (OK OK)
s24-eth3<->s25-eth4 (OK OK)
s25-eth1<->MIT-eth0 (OK OK)
mininet>

```

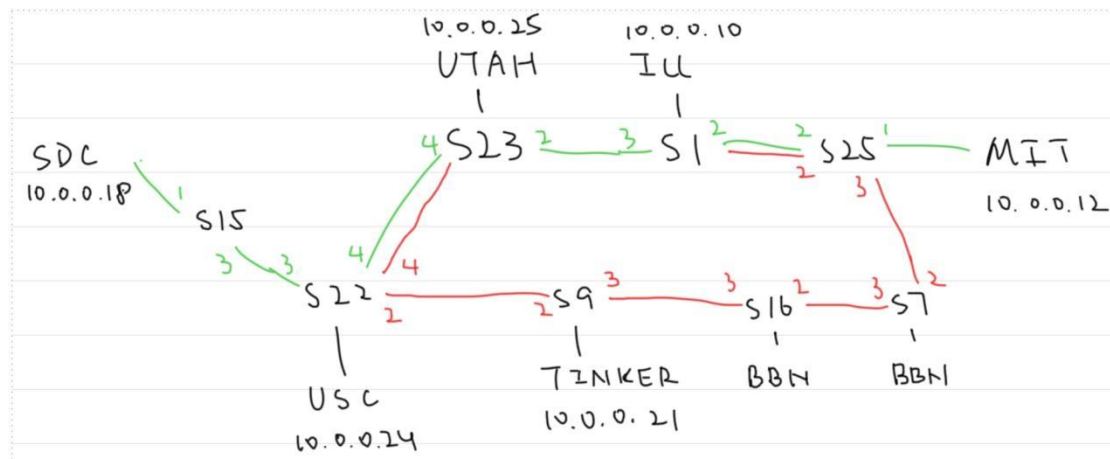
可以看到，MIT 主机连接到交换机 s24 的端口 1，USC 主机连接到交换机 s22 的端口 1。

原本的路径是交换机 s1 连接到交换机 s25 的端口 2，再由交换机 s25 的端口 1 连接到 MIT 主机。

而下发路径后，交换机 s1 连接到交换机 s25 的端口 2，再由交换机 s25 的端口 3 连接到交换机 s7。

可以看到，我们应该分析的关键交换机是 MIT 主机所连接的交换机 s25 与 USC 主机所连接的交换机 s22。

综合上述信息与分析，我们可以用如下的示意图来表示相关连接。



其中，绿色的路径代表原始路径，优先级值为 1。红色的路径为下发路径，优先级值为 10。

在未下发路径时，在 mininet 命令行内输入 `dpctl dump-flows`，查看网络拓扑中各结点的流表。

```
*** s22 -----
-----
cookie=0x0, duration=19.683s, table=0, n_packets=16, n_bytes=960, priority=
65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=7.878s, table=0, n_packets=4, n_bytes=392, priority=1,
ip,in_port="s22-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"
s22-eth4"
cookie=0x0, duration=7.869s, table=0, n_packets=5, n_bytes=490, priority=1,
ip,in_port="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"
s22-eth3"
cookie=0x0, duration=19.683s, table=0, n_packets=14, n_bytes=2041, priority
=0 actions=CONTROLLER:65509
*** s22
```

```
*** s25 -----
-----
cookie=0x0, duration=19.080s, table=0, n_packets=15, n_bytes=900, priority=
65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=7.841s, table=0, n_packets=4, n_bytes=392, priority=1,
ip,in_port="s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"
s25-eth1"
cookie=0x0, duration=7.833s, table=0, n_packets=5, n_bytes=490, priority=1,
ip,in_port="s25-eth1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"
s25-eth2"
cookie=0x0, duration=19.080s, table=0, n_packets=14, n_bytes=1586, priority
=0 actions=CONTROLLER:65509
mininet> █
```


可以看到，交换机 s22 的流表中有 4 个表项，优先级的值分别为 65535，1，1，0。其中 in_port 为 s22-eth3，output 为 s22-eth4 的表项 priority 值为 1。in_port 为 s22-eth4，output 为 s22-eth3 的表项 priority 值为 1。

交换机 s25 的流表中有 4 个表项，优先级的值分别为 65535，1，1，0。其中 in_port 为 s25-eth2，output 为 s25-eth1 的表项 priority 值为 1。in_port 为 s25-eth1，output 为 s25-eth2 的表项 priority 值为 1。

下发路径后，在 mininet 命令行内输入 dpctl dump-flows，查看网络拓扑中各结点的流表。

```
*** s22 -----
----
cookie=0x0, duration=1772.525s, table=0, n_packets=1309, n_bytes=78540, priority=65535, dl_dst=01:80:c2:00:00:0e, dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=1760.720s, table=0, n_packets=4, n_bytes=392, priority=1, ip, in_port="s22-eth3", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
cookie=0x0, duration=1760.711s, table=0, n_packets=5, n_bytes=490, priority=1, ip, in_port="s22-eth4", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s22-eth3"
cookie=0x0, duration=4.469s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s22-eth4", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s22-eth2"
cookie=0x0, duration=4.465s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s22-eth2", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s22-eth4"
cookie=0x0, duration=1772.525s, table=0, n_packets=57, n_bytes=7445, priority=0 actions=CONTROLLER:65509
*** s23 -----
```

```
*** s25 -----
----
cookie=0x0, duration=1771.926s, table=0, n_packets=1308, n_bytes=78480, priority=65535, dl_dst=01:80:c2:00:00:0e, dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=1760.687s, table=0, n_packets=4, n_bytes=392, priority=1, ip, in_port="s25-eth2", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s25-eth1"
cookie=0x0, duration=1760.679s, table=0, n_packets=5, n_bytes=490, priority=1, ip, in_port="s25-eth1", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
cookie=0x0, duration=4.460s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s25-eth3", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s25-eth2"
cookie=0x0, duration=4.456s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s25-eth2", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output:"s25-eth3"
cookie=0x0, duration=1771.926s, table=0, n_packets=56, n_bytes=6920, priority=0 actions=CONTROLLER:65509
mininet> █
```

可以看到，交换机 s22 的流表中有 6 个表项，新增了 2 个表项，优先级的值分别为 65535，1，1，10，10，0，增加了 2 条优先级的值为 10 的流表项。而其中 in_port 为 s22-eth4，output 为 s22-eth2 的表项 priority 值为 10。in_port 为 s22-eth2，output 为 s22-eth4 的表项 priority 值为 10。

交换机 s25 的流表中有 6 个表项，新增了 2 个表项，优先级的值分别为 65535，1，1，10，10，0，增加了 2 条优先级的值为 10 的流表项。其中 in_port 为 s25-eth3，output 为 s25-eth2 的表项 priority 值为 10。in_port 为 s25-eth2，output 为 s25-eth3 的表项 priority 值为 10。

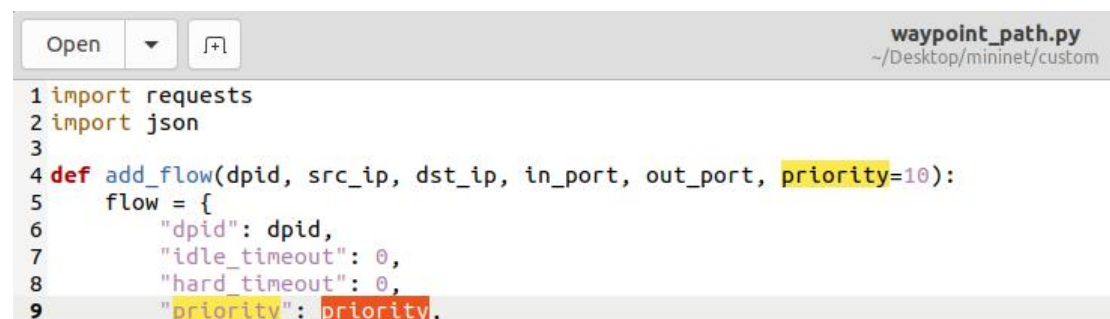
综合交换机 s22 与 s25 的流表与路径进行分析：

SDC 主机向 MIT 主机发送 ping request，数据包由端口 3 送到交换机 s22，根据流表由端口 4 送到交换机 s23，再转发给 s1，s1 转发给交换机 s25 的端口 2。而交换机 s25 中有两条 in_port 为端口 2 的流表项，其中一条为原始路径，通过端口 1 转发给 MIT 主机，优先级为 1。另外一条为下发路径，通过端口 3 转发给 s7，优先级为 10。由于下发路径的流表项优先级更高，会优先匹配这条流表项，所以 ping request 数据包会往 s7 进行转发，而不会发给 MIT 主机，所以无法 ping 通。

而交换机 s7 转发给 s16，s16 转发给 s9，s9 转发给 s22，交换机 s22 的流表项中，in_port 为 2 的流表项转发给端口 4，即 s22 又转发给 s23，这样就形成了一条由原始路径与下发路径共同组成的环路。

7. 拓展实验：修改 waypoint_path.py 中的优先级字段。

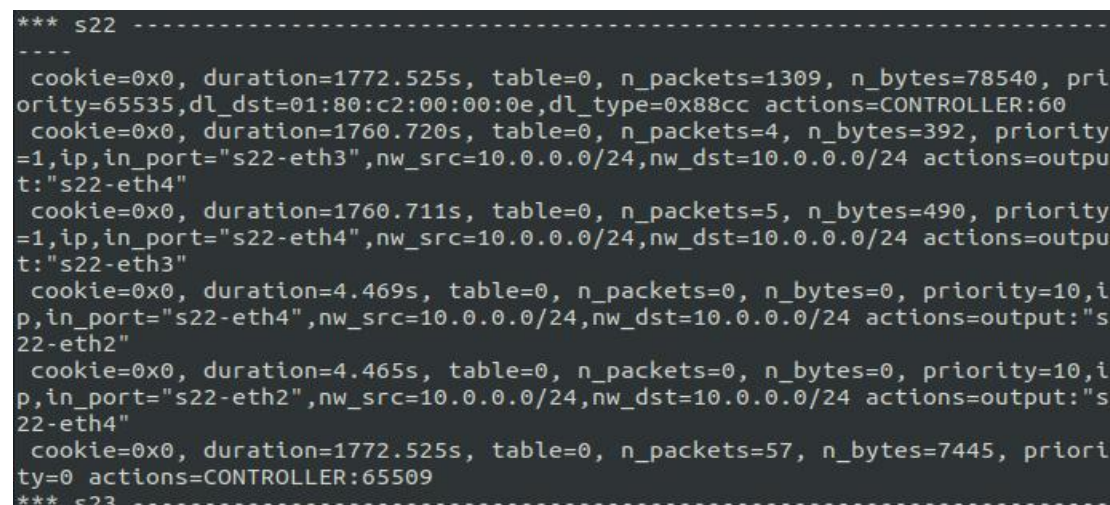
修改 waypoint_path.py 代码中被添加规则的优先级字段，VeriFlow 的检测结果会出错。



```
1 import requests
2 import json
3
4 def add_flow(dpid, src_ip, dst_ip, in_port, out_port, priority=10):
5     flow = {
6         "dpid": dpid,
7         "idle_timeout": 0,
8         "hard_timeout": 0,
9         "priority": priority,
```

在 waypoint_path.py 文件的 add_flow 函数中，如果没有传入指定的 priority 参数，此参数会被默认赋值为 10。

在未修改 priority 默认值时，在 mininet 命令行内输入 dpctl dump-flows，查看网络拓扑中各结点的流表。



```
*** s22 -----
----
cookie=0x0, duration=1772.525s, table=0, n_packets=1309, n_bytes=78540, priority=65535, dl_dst=01:80:c2:00:00:0e, dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=1760.720s, table=0, n_packets=4, n_bytes=392, priority=1, ip, in_port="s22-eth3", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output: "s22-eth4"
cookie=0x0, duration=1760.711s, table=0, n_packets=5, n_bytes=490, priority=1, ip, in_port="s22-eth4", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output: "s22-eth3"
cookie=0x0, duration=4.469s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s22-eth4", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output: "s22-eth2"
cookie=0x0, duration=4.465s, table=0, n_packets=0, n_bytes=0, priority=10, ip, in_port="s22-eth2", nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24 actions=output: "s22-eth4"
cookie=0x0, duration=1772.525s, table=0, n_packets=57, n_bytes=7445, priority=0 actions=CONTROLLER:65509
*** s23 -----
```



```

*** s25 -----
----
 cookie=0x0, duration=1771.926s, table=0, n_packets=1308, n_bytes=78480, pri
ority=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
 cookie=0x0, duration=1760.687s, table=0, n_packets=4, n_bytes=392, priority
=1,ip,in_port="s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:
"s25-eth1"
 cookie=0x0, duration=1760.679s, table=0, n_packets=5, n_bytes=490, priority
=1,ip,in_port="s25-eth1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:
"s25-eth2"
 cookie=0x0, duration=4.460s, table=0, n_packets=0, n_bytes=0, priority=10,i
p,in_port="s25-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s
25-eth2"
 cookie=0x0, duration=4.456s, table=0, n_packets=0, n_bytes=0, priority=10,i
p,in_port="s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s
25-eth3"
 cookie=0x0, duration=1771.926s, table=0, n_packets=56, n_bytes=6920, priori
ty=0 actions=CONTROLLER:65509
mininet>

```

通过之前的分析我们可以看到，形成环路的关键结点是交换机 s22 与 s25。我们还是对这两个交换机流表项进行分析，交换机 s22 的流表中有 6 个表项，优先级的值分别为 65535，1，1，10，10，0。其中 in_port 为 s22-eth4，output 为 s22-eth2 的表项 priority 值为 10。in_port 为 s22-eth2，output 为 s22-eth4 的表项 priority 值为 10。

交换机 s25 的流表中有 6 个表项，优先级的值分别为 65535，1，1，10，10，0。其中 in_port 为 s25-eth3，output 为 s25-eth2 的表项 priority 值为 10。in_port 为 s25-eth2，output 为 s25-eth3 的表项 priority 值为 10。

修改 waypoint_path.py 中的 priority 默认值后，重新启动 Ryu 控制器、启动 VeriFlow 的 Proxy 模式、启动拓扑、建立 SDC 与 MIT 之间的链接、下发路径、再次测试两主机连通性、查看 log_file.txt。

```

sdn@ubuntu:~/Desktop/mininet/custom$ sudo python waypoint_path.py
[sudo] password for sdn:
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
<Response [200]>
install waypoint path: 23 -> 1
23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:s25:2 -> 1

```

在 mininet 命令行内输入 dpctl dump-flows，查看网络拓扑中各结点的流表。


```

*** s22 -----
----
cookie=0x0, duration=66.976s, table=0, n_packets=51, n_bytes=3060, priority
=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=54.857s, table=0, n_packets=9, n_bytes=882, priority=1
,ip,in_port="s22-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:
"s22-eth4"
cookie=0x0, duration=46.021s, table=0, n_packets=0, n_bytes=0, priority=1,i
p,in_port="s22-eth4",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s
22-eth2"
cookie=0x0, duration=46.016s, table=0, n_packets=1410, n_bytes=138180, prio
rity=1,ip,in_port="s22-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=o
utput:"s22-eth4"
cookie=0x0, duration=66.976s, table=0, n_packets=24, n_bytes=3273, priority
=0 actions=CONTROLLER:65509
*** s22

```

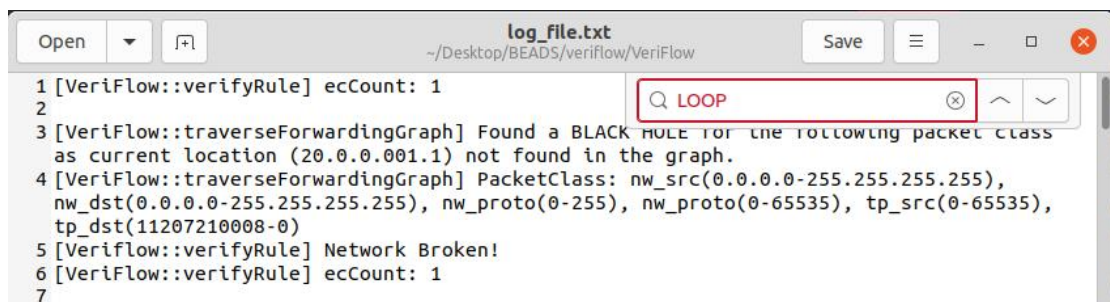
```

*** s25 -----
----
cookie=0x0, duration=66.244s, table=0, n_packets=48, n_bytes=2880, priority
=65535,dl_dst=01:80:c2:00:00:0e,dl_type=0x88cc actions=CONTROLLER:60
cookie=0x0, duration=54.798s, table=0, n_packets=4, n_bytes=392, priority=1
,ip,in_port="s25-eth1",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:
"s25-eth2"
cookie=0x0, duration=45.981s, table=0, n_packets=0, n_bytes=0, priority=1,i
p,in_port="s25-eth3",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=output:"s
25-eth2"
cookie=0x0, duration=45.974s, table=0, n_packets=1416, n_bytes=138768, prio
rity=1,ip,in_port="s25-eth2",nw_src=10.0.0.0/24,nw_dst=10.0.0.0/24 actions=o
utput:"s25-eth3"
cookie=0x0, duration=66.244s, table=0, n_packets=27, n_bytes=3294, priority
=0 actions=CONTROLLER:65509
mininet>

```

可以看到，交换机 s22 的流表中只有 5 个流表项，优先级的值分别为 65535，1，1，1，0。仔细与未下发路径时的流表和未修改 priority 值下发路径后的流表相比较可以发现，缺少了由端口 4 发往端口 3 的流表项。

交换机 s25 的流表中只有 5 个流表项，优先级的值分别为 65535，1，1，1，0。仔细与未下发路径时的流表和未修改 priority 值下发路径后的流表相比较可以发现，缺少了由端口 2 发往端口 1 的流表项。



可以看到，SDC ping MIT 因为存在环路，依旧无法 ping 通，但是此时的 log_file.txt 文件内没有环路路径 LOOP 的记录，VeriFlow 没有检测到环路。

查看 waypoint_path.py 文件中对增加流表项时匹配域的定义。

```

4 def add_flow(dpid, src_ip, dst_ip, in_port, out_port, priority=1):
5     flow = {
6         "dpid": dpid,
7         "idle_timeout": 0,
8         "hard_timeout": 0,
9         "priority": priority,
10        "match":{
11            "dl_type": 2048,
12            "in_port": in_port,
13            "nw_src": src_ip,
14            "nw_dst": dst_ip,
15        },
16        "actions":[
17            {
18                "type":"OUTPUT",
19                "port": out_port
20            }
21        ]
22    }

```

可以看到,增加流表项时,匹配域(match)金由 dl_type, in_port, nw_src, nw_dst 这 4 项, 而 out_port 属于 actions, 不属于匹配域, 不参与匹配。

以交换机 s22 为例, 而未下发路径时, s22 流表中存在由端口 4 发往端口 3 的流表项, in_port 为 4, priority 为 1。而未修改 priority 的值时, 下发路径后, 增加了由端口 4 发往端口 2 的流表项, in_port 也为 4, 但 priority 为 10, 因此可以与上述这条流表项区分开, 不会覆盖这条流表项, 而是新增了一条流表项。

但是修改 priority 值为 1, 下发路径后, 新增的流表项由端口 4 发往端口 2, in_port 也为 4, priority 也为 1, 虽然 out_port 不同, 但是匹配域的 in_port 相同, 因此覆盖了旧的流表项, 而不是新增一条流表项。交换机 s25 的分析也同理, 不再赘述。

即当匹配域相同时, 新流表项覆盖了旧的流表项, 事实上存在环路, 但 VeriFlow 并没有检测出来环路。

而无法发现环路原因是判断环路选择下一跳时, VeriFlow 利用 priority 字段进行了排序, 当出现 priority 相同的规则时, 就会出现这个问题。

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

当出现规则完全匹配需要对原来的规则进行覆盖的时候, VeriFlow 并没有将原来的规则删除并加上新的规则, 而是保留了原来的规则并抛弃了新加的规则。这就导致了 VeriFlow 无法检测到因为下发路径而生成的环路。

8. 拓展实验：域的验证

在 VeriFlow 支持的 14 个域中, 挑选多个域（不少于 5 个）进行验证, 输出并分析结果。

在 EquivalenceClass.h 文件中, 我们可以看到 VeriFlow 支持的域。

```
EquivalenceClass.h
~/Desktop/BEADS/veriflow/VeriFlow

Open  Save  [Menu]

26 enum FieldIndex
27 {
28     IN_PORT, // 0
29     DL_SRC,
30     DL_DST,
31     DL_TYPE,
32     DL_VLAN,
33     DL_VLAN_PCP,
34     MPLS_LABEL,
35     MPLS_TC,
36     NW_SRC,
37     NW_DST,
38     NW_PROTO,
39     NW_TOS,
40     TP_SRC,
41     TP_DST,
42     ALL_FIELD_INDEX_END_MARKER, // 14
43     METADATA, // 15, not used in this version.
44     WILDCARDS // 16
45 };
```

结合原有的 `waypoint_path.py` 文件，我们选择 `in_port`, `dl_src`, `dl_dst`, `dl_type`, `nw_src`, `nw_dst` 这 6 个域进行验证。

恢复基础实验中对 EC 信息打印格式的修改，然后修改 `waypoint_path.py` 如下：

```
1 import requests
2 import json
3
4 def add_flow(dpid, src_ip, dst_ip, in_port, out_port, src_mac, dst_mac, priority=10):
5     flow = {
```

修改了 `add_flow` 函数的定义，增加了两个传入参数 `src_mac` 与 `dst_mac`。

```
5     flow = {
6         "dpid": dpid,
7         "idle_timeout": 0,
8         "hard_timeout": 0,
9         "priority": priority,
10        "match":{
11            "dl_type": 2048,
12            "in_port": in_port,
13            "nw_src": src_ip,
14            "nw_dst": dst_ip,
15            "dl_src": src_mac,
16            "dl_dst": dst_mac
17        },
```

在匹配域中增加了 `dl_src` 与 `dl_dst` 两个匹配域。

```
47 MIT_mac = "00:00:00:00:00:01"
48 SDC_mac = "00:00:00:00:00:02"
49 # send flow mod
50 for node in path:
51     in_port, dpid, out_port = node
52     add_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', in_port, out_port, SDC_mac, MIT_mac)
53     add_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', out_port, in_port, MIT_mac, SDC_mac)
54 show_path(src_sw, dst_sw, path)
```

在 `install_path` 函数中增加了 MIT 与 SDC 的 mac 地址，在添加流表项的时候将这两个 mac 地址作为参数传入。其余部分保存 `waypoint_path.py` 原有代码，不做修改。

修改 `waypoint_path.py` 文件后，重新启动 Ryu 控制器、启动 VeriFlow 的 Proxy 模式、启动拓扑、建立 SDC 与 MIT 之间的链接、下发路径、再次测试两主机连通性、查看 `log_file.txt`。


```

680 [VeriFlow::traverseForwardingGraph] Found a BLACK HOLE for the following packet class
    as current location (20.0.0.023.23) not found in the graph.
681 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
    (1-00:00:00:00:00:01, 1-00:00:00:00:00:01), dl_dst (2-00:00:00:00:00:02,
    2-00:00:00:00:00:02), nw_src (167772416-10.0.1.0, 184549375-10.255.255.255), nw_dst
    (167772160-10.0.0.0, 184549375-10.255.255.255), Field 0 (0, 65535), Field 1 (1, 1),
    Field 2 (2, 2), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0,
    1048575), Field 7 (0, 7), Field 8 (167772416, 184549375), Field 9 (167772160,
    184549375), Field 10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0,
    65535)
682 [VeriFlow::verifyRule] ecCount: 3

```

可以看到，受影响的等价类(EC)个数为 3，即 Field 1, 2 和 3 一共 3 个等价类，与基础实验部分相同。dl_src 为 MIT_mac，dl_dst 为 SDC_mac。

```

684 [VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node
    20.0.0.025.25.
685 [VeriFlow::traverseForwardingGraph] PacketClass: [EquivalenceClass] dl_src
    (2-00:00:00:00:00:02, 2-00:00:00:00:00:02), dl_dst (1-00:00:00:00:00:01,
    1-00:00:00:00:00:01), nw_src (167772160-10.0.0.0, 167772415-10.0.0.255), nw_dst
    (167772160-10.0.0.0, 167772415-10.0.0.255), Field 0 (0, 65535), Field 1 (2, 2), Field 2
    (1, 1), Field 3 (2048, 2048), Field 4 (0, 4095), Field 5 (0, 7), Field 6 (0, 1048575),
    Field 7 (0, 7), Field 8 (167772160, 167772415), Field 9 (167772160, 167772415), Field
    10 (0, 255), Field 11 (0, 63), Field 12 (0, 65535), Field 13 (0, 65535)
686 curr:20.0.0.025.25
687 [VeriFlow::traverseForwardingGraph] Loop path is:
688 20.0.0.025.25 -> 20.0.0.001.1 -> 20.0.0.023.23 -> 20.0.0.022.22 -> 20.0.0.009.9 ->
    20.0.0.016.16 -> 20.0.0.007.7 -> 20.0.0.025.25

```

可以看到，环路路径 LOOP 中，nw_src 与 nw_dst 相同，代表存在环路，且打印出了正确的环路路径：25 -> 1 -> 23 -> 22 -> 9 -> 16 -> 7 -> 25，与设定的环路路径相同。

实验结果：

1. 学习了如何通过下发路径形成转发环路。
2. 学习了如何使用 VeriFlow 检测黑洞和环路。
3. 熟悉了 VeriFlow 的部分源代码和数据结构，自己动手修改源代码完成实验要求。
4. 更加深入地了解了等价类的概念，学习了等价类中包含的信息。
5. 学习了流表中优先级的概念，与添加流表项时可能产生的覆盖。

源代码：

waypoint_path.py

```
import requests
```

```
import json
```

```
def add_flow(dpid, src_ip, dst_ip, in_port, out_port, src_mac, dst_mac,
priority=10):
```

```
    flow = {
```

```
        "dpid": dpid,
```

```
        "idle_timeout": 0,
```

```
        "hard_timeout": 0,
```

```
        "priority": priority,
```

```
        "match":{
```

```

        "dl_type": 2048,
        "in_port": in_port,
        "nw_src": src_ip,
        "nw_dst": dst_ip,
        "dl_src": src_mac,
        "dl_dst": dst_mac
    },
    "actions":[
        {
            "type":"OUTPUT",
            "port": out_port
        }
    ]
}

url = 'http://localhost:8080/stats/flowentry/add'
ret = requests.post(
    url, headers={'Accept': 'application/json', 'Accept':
'application/json'}, data=json.dumps(flow))
print(ret)

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

def install_path():
    '23 -> 4:s22:2 -> 2:s9:3 -> 3:s16:2 -> 3:s7:2 -> 3:25:2 -> 1'
    src_sw, dst_sw = 23, 1
    waypoint_sw = 9 # Tinker 10.0.0.21, s9
    path = [(4, 22, 2), (2, 9, 3), (3, 16, 2), (3, 7, 2), (3, 25, 2)]
    # path = [(3, 7, 2)]
    MIT_mac = "00:00:00:00:00:01"
    SDC_mac = "00:00:00:00:00:02"
    # send flow mod
    for node in path:
        in_port, dpid, out_port = node
        add_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', in_port, out_port,
SDC_mac, MIT_mac)

```

```

        add_flow(dpid, '10.0.0.0/8', '10.0.0.0/8', out_port, in_port,
MIT_mac, SDC_mac)
        show_path(src_sw, dst_sw, path)

if __name__ == '__main__':
    install_path()

```

VeriFlow.cpp

```

bool VeriFlow::verifyRule(const Rule& rule, int command, double& updateTime,
double& packetClassSearchTime, double& graphBuildTime, double& queryTime,
unsigned long& ecCount, FILE* fp)
{
    // fprintf(fp, "[VeriFlow::verifyRule] verifying this rule: %s\n",
rule.toString().c_str());

    .....
    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);
// output to the veriflow console
        fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount); //
output to the FILE *fp, which means the log_file.txt
    }

// check if currentLocation is in path
static bool isInPath(const vector<string>& path, const string& location)
{
    int n = path.size(); //// get the number of nodes on loop
    for(int i =0 ;i < n;i++) // go through the path
    {
        if(path[i] == location)
        {
            return true;
        }
    }
}

```



```

        return false;
    }

    // print loop and infomation
    static void printLoop(FILE* fp, const vector<string>& path, const string&
current)
    {
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");
        int n = path.size(); // get the number of nodes on loop
        for(int i = 0; i < n; i++)
        {
            fprintf(fp, "%s -> ", path[i].c_str()); // print path nodes
        }
        fprintf(fp, "%s\n", current.c_str()); // print current node
    }

    // modify the arguments
    bool VeriFlow::traverseForwardingGraph(const EquivalenceClass&
packetClass, ForwardingGraph* graph, const string& currentLocation, const
string& lastHop, unordered_set< string > visited, vector<string>& path,
FILE* fp)
    {
        .....
        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());
            //printloop
            fprintf(fp, "curr:%s\n", currentLocation.c_str());
            printLoop(fp, path, currentLocation);
            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);
    }

```

```

        if(!isInPath(path, currentLocation))
        {
            path.push_back(currentLocation); // If not, add currentLocation to
path
        }

```

EquivalenceClass.cpp

// Modify the toString method of EquivalenceClass

```

string EquivalenceClass::toString() const
{
    char buffer[1024];
    sprintf(buffer, "nw_src(%s-%s), nw_dst(%s-%s), nw_proto(%lu-%lu),
nw_proto(%lu-%lu), tp_src(%lu-%lu), tp_dst(%lu-%lu)",
        ::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(),
        this->lowerBound[NW_PROTO],
        this->upperBound[NW_PROTO],
        this->lowerBound[TP_SRC],
        this->upperBound[TP_SRC],
        this->lowerBound[TP_DST],
        this->upperBound[TP_DST]);
    string retVal = buffer;
    return retVal;
}

```