SDN 第四次实验报告

计算机 71　　王丹　　2171321099

一、实验内容

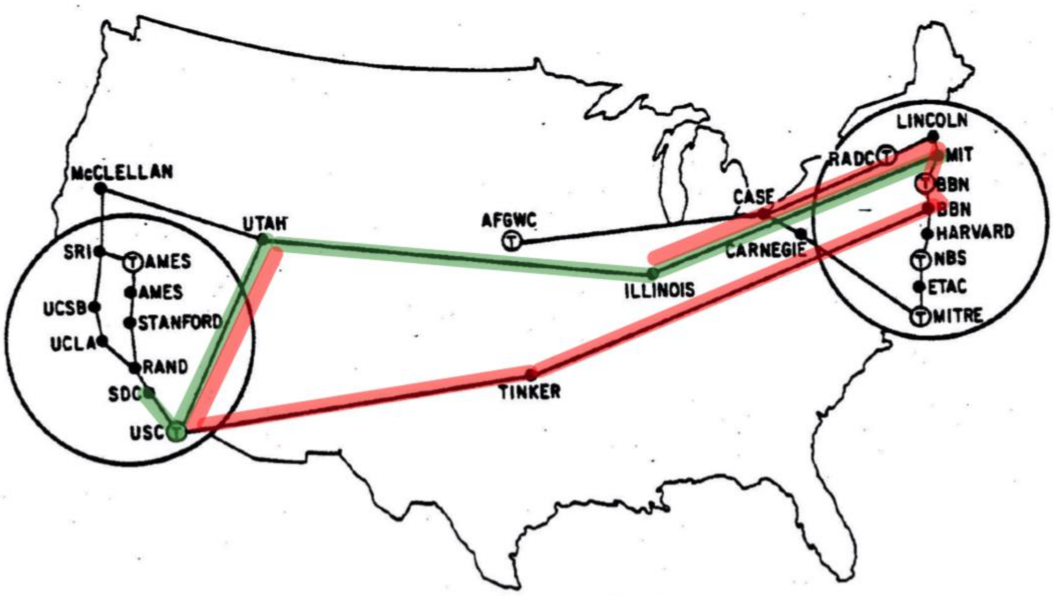
假如你是生活在1972年维护ARPAnet的网络管理员，在前面的实验中你学会了如何建立最短路径，下发了一条SDC到MIT跳数最少的路径（图中绿色的路径）。你的同事Bob某天接到了一个新的需求，要求UTAH到ILLINOIS之间的所有流量必须经过部署于TINKER的流量分析器以进行进一步研究，粗心大意的Bob没有检查当前的网络状态就很快下发了一条新的路径（图中红色的路径）。聪明又机智的你很快意识到Bob下发的流表很可能造成转发的环路。

现要求你运行VeriFlow工具，对上述两条转发路径进行检查，完成下面的任务：

1. 输出每次影响EC的数量
2. 打印出环路路径的信息
3. 进一步打印出环路对应的EC的相关信息
4. 分析原始代码与补丁代码的区别，思考为何需要添加补丁

在完成以上任务的基础上，另有下面的选作任务：

1. 若修改`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，试描述错误是什么，并解释出错的原因
2. 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



二、解决方案

1. 输出每次影响的 EC 的数量

   在 verifyRule()函数中，通过调用 getAffectedEquivalenceClasses()函数得到受影响的 EC 并将其保存在 vFinalPacketClasses 中，因此，vFinalPacketClasses 中包含的 EC 的数量即为受影响的 EC 的数量

   在 verifyRule()函数中相应位置加上下图框中的代码即可输出每次影响的 EC 的数量

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

2. 打印出环路路径的信息

在 verifyRule()函数中，通过调用 traverseForwardingGraph()函数来遍历指定 EC 的转发图，检验其中是否有环路（loop）或黑洞（black hole）。用 visited 集合来保存已经遍历过的结点，如当前结点已在 visited 中出现过，则说明转发图中存在环路。原始代码中 visited 的类型为 unordered_set<string>，而这是一种无序的数据结构，只能保留结点是否已被遍历的信息而无法保留结点遍历顺序的信息。因此将 visited 的类型改为 vector<string>。当检测到环路时，将 visited 中的结点依次输出即得到环路路径。

在 traverseForwardingGraph()函数中加入下图框中的代码即可（同时，因为将 visitied 由 unordered_set<string>改为 vector<string>，代码中相应的有多处需要改动的地方，如 find 和 insert，以及 veriflow.h 中的函数声明）。

```
if(find(visited.begin(), visited.end(), currentLocation) != visited.end())
{
        // Found a loop.
        fprintf(fp, "\n");
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node %s.\
        //fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toString().c_str());
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] PacketClass: %s\n", packetClass.toShortString().c_str())
        fprintf(fp, "[VeriFlow::traverseForwardingGraph] Loop path is:\n");
        vector< string >::iterator cur = find(visited.begin(), visited.end(), currentLocation);
        vector< string >::iterator it = cur;
        fprintf(fp, "%s", (*it).c_str());
        it++;
        for(; it != visited.end(); it++) {
                fprintf(fp, " -> %s", (*it).c_str());
        }
        fprintf(fp, " -> %s\n", (*cur).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;
}
```

解释：首先在 visited 中找到 currentLocation，然后从相应位置开始遍历 visited 直到末尾

3. 打印出环路对应的 EC 的相关信息

EquivalenceClass 中的匹配域的定义如下：

```
enum FieldIndex
{
        IN_PORT, // 0
        DL_SRC,
        DL_DST,
        DL_TYPE,
        DL_VLAN,
        DL_VLAN_PCP,
        MPLS_LABEL,
        MPLS_TC,
        NW_SRC,
        NW_DST,
        NW_PROTO,
        NW_TOS,
        TP_SRC,
        TP_DST,
        ALL_FIELD_INDEX_END_MARKER, // 14
        METADATA, // 15, not used in this version.
        WILDCARDS // 16
};
```

各个域的匹配范围是通过 lowerBound 和 upperBound 来界定的

例如 lowerBound[NW_SRC] – upperBound[NW_SRC]为 NW_SRC 域的匹配范围

按照题目要求提取 TCP/IP 五元组作为主要信息显示，即提取 NW_SRC, NW_DST, NW_PROTO, TP_SRC, TP_DST 五个域的信息，代码如下：

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

        string retVal = buffer;
        retVal += ", ";

        sprintf(buffer, "nw_proto (%lu-%lu), tp_src (%lu-%lu), tp_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;
}
```

在 EquivalenceClass 中添加新函数 toShortString()，该函数只提取 TCP/IP 五元组的信息来生成字符串并返回，然后在 VeriFlow 中的 traverseForwardingGraph()函数中的相应位置用对 toShortString()的调用替换对 toString()的调用即可

4. 分析原始代码与补丁代码的区别，思考为何要添加补丁

通过阅读 0001-for-xjtu-sdn-exp-2020.patch 文件发现，补丁代码主要做出了以下两个方面的改动：

1) 弃用 Rule 的 IN_PORT 域，为 Rule 类增加 in_port 属性
2) 在遍历转发图时通过考虑上一跳来发现一种新的黑洞（black hole）类型。源代码中只有两种类型的 black hole，一种是转发图中没有当前结点；一种是转发图中有当前结点而当前结点处无转发规则；新的 black hole 类型为：转发图中有当前结点且当前结点处有转发规则却无法匹配（上一跳的 IP 地址为 lastHop，而当前结点处全部转发规则的上一跳均无法匹配 lastHop）

通过分别运行打补丁前后的代码，可以发现打上补丁之后，VeriFlow 发现了第三种类型的 black hole

```
[VeriFlow::traverseForwardingGraph] ***Found a BLACK HOLE for the following packet class as there is no outgoing link at
[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

[VeriFlow::traverseForwardingGraph] ***Found a BLACK HOLE for the following packet class as there is no outgoing link at
[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

[VeriFlow::traverseForwardingGraph] ***Found a BLACK HOLE for the following packet class as there is no outgoing link at
[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
```

***表示为第三种类型的 black hole

5. 选做 1——试修改 waypoint_path.py 代码中被添加规则的优先级字段，将 10 修改为 1，VeriFlow 的检测结果会出错，试描述错误是什么，并解释出错的原因

错误描述：VeriFlow 未检测出环路，而 SDC 和 MIT 无法 ping 通

出错原因：

当有新规则要下发时，将新规则加入到规则树中的代码执行流程如下：

OpenFlowProtocolMessage::process()→OpenFlowProtocolMessage::processFlowMod()→VeriFlow::verifyRule()→VeriFlow::getAffectedEquivalenceClasses()→VeriFlow::addRule()→Trie::addRule()

在 Rule 类中，equals()函数如下：

```
bool Rule::equals(const Rule& other) const
{
	for(int i = 0; i < ALL_FIELD_INDEX_END_MARKER; i++)
	{
		if((this->fieldValue[i].compare(other.fieldValue[i]) != 0)
				|| (this->fieldMask[i].compare(other.fieldMask[i]) != 0))
		{
			return false;
		}
	}

	if((this->type == other.type)
			&& (this->wildcards == other.wildcards)
			&& (this->location.compare(other.location) == 0)
			&& (this->in_port == other.in_port)
			// && (this->nextHop.compare(other.nextHop) == 0) // Not present in OFPT_FLOW_REMOVED messag
			&& (this->priority == other.priority)
			// && (this->outPort == other.outPort) // Not used in this version.
	)
	{
		return true;
	}
	else
	{
		return false;
	}
}
```

可以看出，将被添加规则的优先级改为 1 后，新添加的规则和旧规则被判断为相等（新旧规则只有 nextHop 不同，而判定两个 rule 是否相等时却没有考虑 nextHop），因此新规则并未被添加到规则树中，而观察交换机上的流表项发现，新规则覆盖了旧规则

```
[VeriFlow::verifyRule] verifying this rule:
[VeriFlow::addRule] rule already in trie
old rule: [Rule] type: 1, dlSrcAddr: 00:00:0
new rule: [Rule] type: 1, dlSrcAddr: 00:00:0
[VeriFlow::verifyRule] ecCount: 3
```

```
location: 20.0.0.22, nextHop: 20.0.0.15, in_port: 4, priority: 1,
location: 20.0.0.22, nextHop: 20.0.0.9, in_port: 4, priority: 1, w
```

新旧规则只有 nextHop 不一样，但是 nextHop 不参与比较两条规则是否一致

未修改优先级时：

```
sudo ovs-ofctl dump-flows s22
91s, table=0, n_packets=154, n_bytes=9240, priority=65535,dl_dst=01:80:c2:00:00:0e,dl_type
55s, table=0, n_packets=18, n_bytes=1764, priority=1,ip,in_port="s22-eth3",nw_src=10.0.0.0
51s, table=0, n_packets=9, n_bytes=882, priority=1,ip,in_port="s22-eth4",nw_src=10.0.0.0/2
2s, table=0, n_packets=0, n_bytes=0, priority=10,ip,in_port="s22-eth4",nw_src=10.0.0.0/24
8s, table=0, n_packets=3110, n_bytes=304780, priority=10,ip,in_port="s22-eth2",nw_src=10.0
45s, table=0, n_packets=43, n_bytes=4326, priority=0 actions=CONTROLLER:65509
```

修改优先级后:

```
sudo ovs-ofctl dump-flows s22
67s, table=0, n_packets=102, n_bytes=6120, priority=65535,dl_dst=01:80:c2:00:00:0e,d
29s, table=0, n_packets=18, n_bytes=1764, priority=1,ip,in_port="s22-eth3",nw_src=10
3s, table=0, n_packets=0, n_bytes=0, priority=1,ip,in_port="s22-eth4",nw_src=10.0.0.
3s, table=0, n_packets=1641, n_bytes=160818, priority=1,ip,in_port="s22-eth2",nw_src=
90s, table=0, n_packets=42, n_bytes=4256, priority=0 actions=CONTROLLER:65509
```

也就是说，在交换机中，新的流表项覆盖了旧的流表项，而在 VeriFlow 维护的 trie 中，保留了旧流表项而未加入新的流表项，所以 VeriFlow 并未检测出环路，但是 MIT 和 SDC 却无法 ping 通

因此，在检测出 rule already in trie 时，删除旧规则，加入新规则，即可解决问题。

```
else
{
        unordered_set< Rule, KHash< Rule >, KEqual< Rule > >::const_iterator itr;
        itr = leaf->ruleSet->find(rule);
        if(itr != leaf->ruleSet->end()) // Rule already exists.
        {
                fprintf(logFile, "[VeriFlow::addRule] rule already in trie\nold rule: %s\nnew rule: %s\n", itr->toString
                leaf->ruleSet->erase(itr);
                //return false;
        }
}

leaf->ruleSet->insert(rule);
```

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAdd
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
```

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAdd
[VeriFlow::addRule] rule already in trie
old rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:
new rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

6. 选做 2——在 VeriFlow 支持的 14 个域中，挑选多个域（不少于 5 个）进行验证

1	8	16	24 32
version	type	length	
xid			
wildcard			
in_port		dl_src	
dl_src			
dl_dst			
dl_dst		dl_vlan	
dl_vlan_pcp	pad	dl_type	
nw_tos	nw_proto	pad	
nw_src			
nw_dst			
tp_src		tp_dst	
cookie			
cookie			
command		idle_time	
hard_time		priority	
buffer_id			
out_port		flags	
actions[0]			

```
enum FieldIndex
{
        IN_PORT, // 0
        DL_SRC,
        DL_DST,
        DL_TYPE,
        DL_VLAN,
        DL_VLAN_PCP,
        MPLS_LABEL,
        MPLS_TC,
        NW_SRC,
        NW_DST,
        NW_PROTO,
        NW_TOS,
        TP_SRC,
        TP_DST,
        ALL_FIELD_INDEX_END_MARKER, // 14
        METADATA, // 15, not used in this version.
        WILDCARDS // 16
};
```

可以看出，FLOW_MOD 消息可以支持的匹配域和 VeriFlow 支持的验证域的交集为：IN_PORT, DL_SRC, DL_DST, DL_TYPE, DL_VLAN, DL_VLAN_PCP, NW_SRC, NW_DST, NW_PROTO, NW_TOS, TP_SRC, TP_DST

挑选 in_port, dl_type, nw_src, nw_dst, nw_proto, dl_vlan, dl_vlan_pcp, tp_src, tp_dst 等域来进行验证

取值：

① 因为实验中主要用 ping 命令测试 MIT 和 SDC 之间的连通性，使用到的协议为 ICMP，因此 dl_type=2048, nw_proto=1

② 因为发现网络拓扑中主机的 IP 地址都在 10.0.0.1-10.0.0.25 的范围内，因此 nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

③ dl_vlan, dl_vlan_pcp, tp_src, tp_dst 的值都是随意指定的

1) waypoint_path.py：

   dl_type=2048, nw_proto=1, nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

   shortest_path.py：

   dl_type=2048, nw_proto=1, nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

   预期结果：同之前一致

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAdd
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

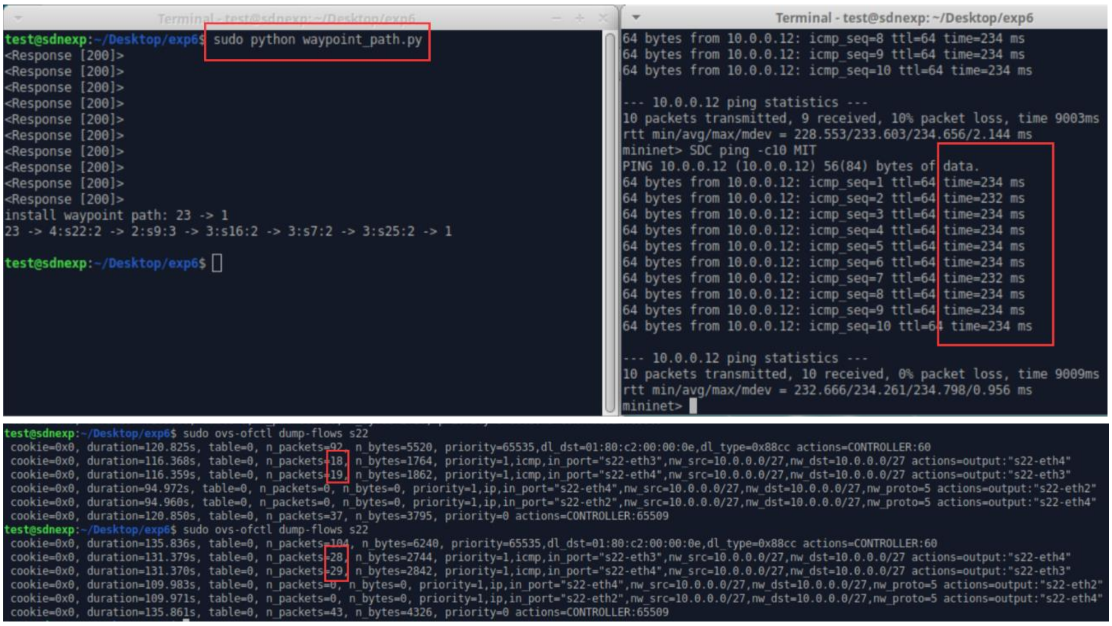
2) waypoint_path.py：

   dl_type=2048, nw_proto=5, nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

   shortest_path.py：

   dl_type=2048, nw_proto=1, nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

   预期结果：不会造成环路，SDC ping MIT 还能 ping 通



执行完 waypoint_path.py 之后，SDC 和 MIT 依旧可以 ping 通，且时延与之前一致，icmp 数据包匹配的仍然是旧流表项，说明新规则的下发并没有对 ping 包的转发路径造成影响

3) waypoint_path.py:

   dl_type=2048, nw_proto=1, nw_src=10.0.0.0/27, nw_dst=10.0.0.0/27

   shortest_path.py：

   dl_type=2048, nw_proto=1, nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24

   预期结果：执行 waypoint_path.py 之后，根据 ip 的最长匹配原则，数据包会匹配 10.0.0.0/27 的流表项，从而走入环路，SDC 和 MIT 之间无法 ping 通

   实际结果：数据包匹配旧流表项，SDC 和 MIT 之间依旧可以 ping 通（好像和 OVS 的缓存机制有关）

但是在 logfile 中可以看到 veriflow 依旧正确检测出环路：

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAddr
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAddr
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.31), nw_dst (10.0.0.0-10.0.0.31), nw_proto (1-1), tp_src (0-65535), tp_dst(0-65535)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

4) waypoint_path.py：

dl_type=2048, nw_proto=1, nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24, dl_vlan=7,dl_vlan_pcp=7, tp_src=100, tp_dst=100

shortest_path.py：

dl_type=2048, nw_src=10.0.0.0/24, nw_dst=10.0.0.0/24

预期结果：执行 waypoint_path.py 之后 SDC 和 MIT 之间依旧可以 ping 通，veriflow 检测出环路

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAddrMa
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDstAddrMa
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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 (1-1), tp_src (100-100), tp_dst(100-100)
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

```
, dl_type: 2048, dl_vlan: 7, dl_vlan_pcp: 7, mpls_label: 0, mpls_tc: 0, nw_proto: 1, nw_tos: 0, tp_src: 100, tp_dst: 100
```

三、实验结果

1. 1,2,3 问结果

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDst
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
```

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlDstAddr: 00:00:00:00:00:00, dlDst
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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), tp_src (0-65535), tp_dst(
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

2. 选做 1 结果

   正确结果（priority=10）：

```
[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

   错误结果（priority=1）：

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0, dlD
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.23.
[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-

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.23.
[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-

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.23.
[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-
```

四、实验中的一些发现：

1. 阅读源代码的过程中发现，VeriFlow 只是对新规则进行验证，而不会进行主动干预。例如，新的规则会造成环路，VeriFlow 只会输出相应的提示信息，而不会阻止该规则的下发。与助教交流后也验证了这个观察，VeriFlow 是比较早期的文章，主要贡献在于提出了 EC 的思想完成了数据平面的实时性检验，后来有一些新的工作就是基于 VeriFlow 的，比如 NSDI'18 的 NEAt 就在 VeriFlow 上完成了进一步的修复工作
2. 在完成选做 1 时，一开始尝试了另一种解决方案

   在比较两条规则是否相等时考虑 nextHop，并且为 Rule 添加 timestamp 属性用以对规则进行排序

将&& (this->nextHop.compare(other.nextHop) == 0)取消注释之后，VeriFlow 又可以正常检测出环路，但是只检测出一个方向的环路，未检测出另一个方向的环路是因为在 s25，in_port=2 时有两条优先级相同的转发规则可以匹配，一条是旧规则，nextHop=10.0.0.12，一条是新规则，nextHop=20.0.0.7，遍历转发图时执行旧规则，所以日志文件中输出 The following packet class reached destination at node 20.0.0.25

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.

[VeriFlow::traverseForwardingGraph] The following packet class reached destination at node 20.0.0.25.
[VeriFlow::traverseForwardingGraph] PacketClass: nw_src (10.0.0.0-10.0.0.255), nw_dst (10.0.0.0-10.0.0.
```

如果为规则加上时间戳，并在对规则进行排序时，优先按照优先级排序，优先级一致时按照时间戳排序，便可解决这一问题：
为 Rule 添加 timestamp 属性后两个方向的环路都可以正常被检测出来：

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0,
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.1 -> 20.0.0.23 -> 20.0.0.22 -> 20.0.0.9 -> 20.0.0.16 -> 20.0.0.7 -> 20.0.0.25
```

```
[VeriFlow::verifyRule] verifying this rule: [Rule] type: 1, dlSrcAddr: 00:00:00:00:00:00, dlSrcAddrMask: 0:0:0:0:0:0,
[VeriFlow::verifyRule] ecCount: 3

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25

[VeriFlow::traverseForwardingGraph] Found a LOOP for the following packet class at node 20.0.0.25.
[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
[VeriFlow::traverseForwardingGraph] Loop path is:
20.0.0.25 -> 20.0.0.7 -> 20.0.0.16 -> 20.0.0.9 -> 20.0.0.22 -> 20.0.0.23 -> 20.0.0.1 -> 20.0.0.25
```

添加的代码如下：

```
--- a/veriflow/VeriFlow/Rule.h
+++ b/veriflow/VeriFlow/Rule.h
@@ -19,6 +19,7 @@
 #include <string>
 #include "EquivalenceClass.h"
 #include "EquivalenceRange.h"
+#include <time.h>

 using namespace std;

@@ -43,6 +44,7 @@ public:
            string nextHop;
            unsigned int in_port;
            uint16_t priority;
+           time_t timestamp;
            // uint16_t outPort; // Not used in this version.
```

```
@@ -34,6 +34,7 @@ Rule::Rule()

            this->wildcards = 0;
+           this->timestamp = 0;
            this->location = "";
            this->nextHop = "";
            this->in_port = 65536;
@@ -53,6 +54,7 @@ Rule::Rule(const Rule& other)

            this->wildcards = other.wildcards;
+           this->timestamp = other.timestamp;
            this->location = other.location;
            this->nextHop = other.nextHop;
            this->in_port = other.in_port;
```

```
--- a/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
+++ b/veriflow/VeriFlow/OpenFlowProtocolMessage.cpp
@@ -17,6 +17,7 @@
 #include "openflow.h"
 #include "Network.h"
 #include "VeriFlow.h"
+#include <time.h>

 void OpenFlowProtocolMessage::process(const char* data, ProxyConnectionInfo& info, FILE* fp)
 {
@@ -406,6 +407,7 @@ void OpenFlowProtocolMessage::processFlowMod(const char* data, ProxyConnectionIn
                                        Rule rule;
                                        rule.type = FORWARDING;
                                        rule.wildcards = ntohl(ofm->match.wildcards);
+                                       rule.timestamp = time(NULL);

                                        rule.fieldValue[IN_PORT] = "0";
                                        rule.fieldMask[IN_PORT] = "0";//((rule.wildcards == OFPFW_ALL) || ((rul
```

```
--- a/veriflow/VeriFlow/VeriFlow.cpp
+++ b/veriflow/VeriFlow/VeriFlow.cpp
@@ -452,7 +452,11 @@ string convertIntToString(unsigned int value)

 bool compareForwardingLink(const ForwardingLink& first, const ForwardingLink& second)
 {
-           if(first.rule.priority >= second.rule.priority)
+           if(first.rule.priority > second.rule.priority)
+           {
+                       return true;
+           }
+           else if(first.rule.priority == second.rule.priority && first.rule.timestamp > second.rule.timestamp)
            {
                        return true;
            }
```

这种解决方案虽然将新规则加入到了规则树中，且实现了优先匹配新规则，但规则树中还保留了旧规则。而 VeriFlow 正确工作的前提应该是 Trie 中的规则和交换机中的流表项一致，不能多不能少，因此这种解决方案不可取。但是为规则添加时间戳，并且在对规则进行排序时，优先根据优先级排序，若优先级相同则按照时间戳排序，可以保证在遍历转发图时，若两条规则的匹配字段相同，且优先级一致，则根据最新的规则确定下一跳，这种思想值得被记录。

五、实验心得

此次实验的 1、2、3 问都比较简单，属于确定结果的题目，而 4、5、6 问则类似于开放性题目，要求我们做出更多的思考。做实验的时候，1、2、3 问很快就做出来了，但是 4、5 问却做了两天的时间，从一开始的毫无头绪，到后来的渐渐明朗，是通过不断的观察取得的，一开始没有思路不要紧，把能观察到的实验现象都记录下来，等积累到一定程度，不同实验现象之间的联系便开始显现出来。并且，5 问并没有要求给出问题的

解决方案，但是问题明确之后，解决方案也是显而易见的，虽然一开始走了一点弯路，但也收获了很多。通过此次实验，不仅对 veriflow 的原理及实现有了更深的认识，而且也熟练了对 git 的使用，收获颇丰。

六、实验过程记录

在完成此次实验的过程中，我通过撰写 README 文档的形式记录下了完整的实验过程。部分截图如下：

```
1.update the PATH environment variable to include the VeriFlow build directory
$su
#vi /etc/environment
add :/home/test/BEADS/veriflow/VeriFlow to PATH
#source /etc/environment

---------------------------------------------------------------------------------------------------------

2.print ecCount
add the following codes in function verifyFlow in VeriFlow.cpp:
1)
        fprintf(fp, "[VeriFlow::verifyRule] verifying this rule: %s\n", rule.toString().c_str());
2)
        if(ecCount == 0)
        {
                ...
        }
        else
        {
                fprintf(fp, "[VeriFlow::verifyRule] ecCount: %lu\n", ecCount);
        }

---------------------------------------------------------------------------------------------------------

3.modify EC format
1)
add function toShortString() in EquivalenceClass.cpp
string EquivalenceClass::toShortString() const
{
        char buffer[1024];
        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());

        string retVal = buffer;
        retVal += ", ";

        sprintf(buffer, "nw_proto (%lu-%lu), tp_src (%lu-%lu), tp_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;
```