University of Amsterdam

MSc Security and Network Engineering
Research Project 2

Improving availability in Industrial Control Systems using Software-Defined Networking

February 7, 2021

Students:
Marios Andreou
marios.andreou@os3.nl

Joris Jonkers Both
joris.jonkersboth@os3.nl

Supervisors:
Pavlos Lontorfos
Dominika Rusek

Lecturer:
Cees de Laat

Abstract

This research aims to improve availability in Industrial Control Systems (ICS) using Software-Defined Networking (SDN). Programmable Logic Controllers (PLC) are the brains of ICSs. If these critical devices would become unavailable, it could lead to operational disruptions and widespread damage. Network Function Virtualization (NFV) could be used to create a more agile and lower cost infrastructure, since hardware could be virtualized. On top of that, Software-Defined Networking (SDN) can be used to manage networks by separating the control plane from the data plane. Our research has shown that SDN combined with NFV provides efficiency, flexibility and reduces human error since the action taken by the software will be dynamic. This raises the question whether SDN combined with NFV could enhance the availability of ICS environments and thus reduces the risk of disruptions and widespread damage. According to previous research, on average there is a downtime of one hour, since it will take 30 minutes to summon a technician and another 30 minutes to repair the faulty network module (e.g. switch). This research has tried to answer this question by implementing three SDN driven scenarios: re-routing of the traffic, redeployment of network hardware and recreation of specific interfaces of the network hardware. Our experiments showed that all three scenarios could be used to improve the availability of ICS environments. However, differences can be observed between these scenarios. Re-routing traffic in case of a switch failure showed to perform the worst which had an average downtime of 5576ms, while the scenario in which a switch would be redeployed showed to perform the best which had an average downtime of only 48ms. However, we were unable to obtain 100% availability. Since the NFV function was written in Python, the efficiency of the code did not allow us to set an interval small enough to detect failures fast enough. Moreover, the performance of the machine the code was running on, could also have played a role. Using SDN and NFV will result in complexity in an ICS environment. One would trade simplicity for the speed of recovery in case of a hardware failure, availability and also the ability of having more machines running while using less hardware.

Keywords: Software-Defined Networking (SDN), Industrial Control Systems (ICS), Network Function Virtualization (NFV)

1 Introduction

Programmable Logic Controllers (PLC) are the brains of Industrial Control Systems (ICS). If these critical devices would become unavailable, it could lead to operational disruptions and loss of money [25]. In recent years, attacks on ICSs have risen dramatically since nowadays these systems are often IP-based and connected to other networks, or even to the Internet [7]. With these attacks, malicious external actors seek to compromise or steal information from the underlying technologies in industrial processes, such as critical controllers [2].

However, cyber attacks are not the only factors that could lead to a threat in an ICS infrastructure. Hardware failures can also contribute to significant downtime. Due to the fact that programmable logic controllers need to have a high availability, the network infrastructure needs to be resilient against disruption. One of the technologies that can be used for this purpose is Network Function Virtualization (NFV) [11]. This technology allows one to virtualize network equipment and therefore create a more agile, lower cost network infrastructure. Moreover, automation reduces the amount of human intervention, which allows for autonomous error handling (e.g. if a network switch fails).

Furthermore, Software Defined Networking (SDN) can be used to manage networks by separating the control plane from data plane. This makes it easier for network engineers to monitor and configure traffic streams, diagnose threats, as well as apply, modify or remove security policies through a centralized or distributed controller [26]. Since SDN combined with NFV provides efficiency and flexibility by optimizing existing applications, services and infrastructure, it could allow for a more reliable ICS environment [27]. For this project we will be investigating the appropriate methods to minimize the risk of downtime in an ICS environment using NFV and SDN. This will give a better understanding of how SDN could be applied in ICS environments.

The main research question for this project is defined as follows:

How could Software Defined Networking combined with Network Function Virtualization enhance availability in an Industrial Control System in case of a network hardware failure?

In order to answer our research question, the following sub-questions will be answered:

- How can SDN combined with NFV provision back-up network equipment to maintain availability during a network failure?
- What are the consequences of provisioning back-up network equipment in an ICS environment for the manageability and connectivity of the network and its connected PLCs?
- What are the limitations of using SDN combined with NFV in an ICS environment regarding the availability of the connected PLCs?

2 Related Work

There has been some research done into the implementation of SDN and NFV in an ICS environment. Zhou et. al. showed in 2017 that SDN technology combined with NFV technology can be used in ICS environments to detect and mitigate DDoS attacks by scaling up resources and/or updating data forwarding rules [27].

On top of that, in the same year, Piedrahita et. al. showed that SDN technology can be used to respond to security incidents and vulnerabilities by discarding packets from or to a specific device on the network [21]. To accomplish this, an SDN system and an IDS

could complement each other, in which the IDS would notify SDN about every incident or vulnerability. Also, Chavez researched SDN to find out if Moving Target Defense, which is used to confuse an attacker by changing the IP address of a system as soon as an adversary has discovered an IP address, could be effective in securing ICS environments [6].

More recent, in 2019, Brugman et al did propose a new architecture in which a Cloud Based Intrusion Detection and Prevention System is used. This technique leverages SDN as mechanism to send network traffic to the cloud for analysis and decision making. Furthermore, some research has been done regarding the implementation of SDN in industrial automation. Ahmed et. al. showed that SDN can be used in industrial automation to improve the efficiency and productivity while reducing the human intervention required [1]. Next, a research has determined that on average it would take 30 minutes to summon a technician and another 30 minutes to fix a network component in a virtualized environment [15].

Moreover, Kálmán also showed that SDN in industrial operations can be applied for traffic segmentation and security measures [13]. Finally, a research done in 2020 has shown that SDN can be used to implement hardware failover in an ICS environment [17]. However, the focus of that research was not on automatic redeployment of the faulty hardware via SDN, but instead on how SDN can help regarding redundancy, scalability and security.

These researches all focused on how SDN could help mitigate or detect attacks and improve efficiency, productivity and security on an ICS system. However, we were unable to find any research that was done into how SDN combined with NFV could benefit in terms of availability in an ICS environment yet.

3 Background

3.1 Software-Defined Networking

Software-Defined Networking (SDN) is a new approach to communication networks which, through abstraction, tries to eliminate the limitations, like having to manually configure every network device individually, that traditional networking has. SDN can achieve this, by introducing a centralized control logic and separating control plane, data plane and management plane as illustrated in Figure 1. Starting from bottom-up, in the data plane we have our network infrastructure which consists of forwarding hardware (e.g. switches) and a southbound interface which is responsible for establishing communication between the data plane and control plane (e.g. OpenFlow). In the control plane, we have the Network Operating System (NOS), i.e. controller(s), which is considered the brain of SDN and therefore, it's responsible for the abstraction from the hardware, traffic routes, network topology, state details, statistic details and more. Moreover, a northbound interface is required in order to connect the control plane and management plane (e.g. REST, RESTful). Finally, in the management plane, we can visualize the data gathered by the controller using application layer protocols, which gives the ability and flexibility to manage the network [8].

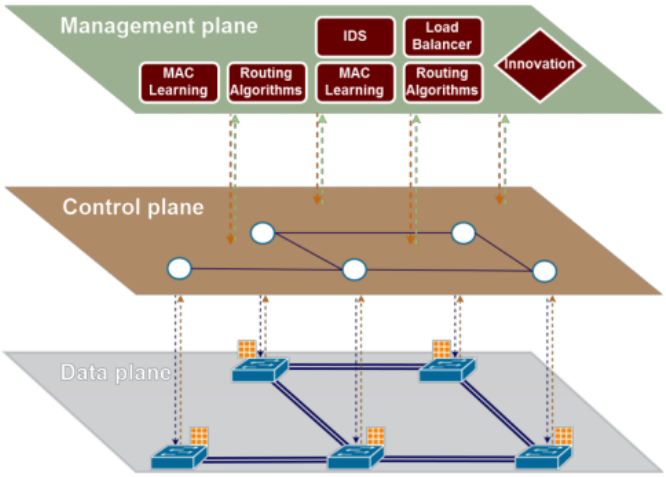

Figure 1: SDN Architecture. [8]

3.2 OpenFlow

As discussed before, SDN uses north- and southbound interfaces. One of the standards for southbound interfaces is OpenFlow. It was developed by Stanford University in 2008. Since then, it has become the most widely used standard for southbound interfaces [19]. It is designed to standardize the communication between the control plane and the data plane [16]. OpenFlow can be used to program the flow table of different switches. A flow table contains a sequence of rules, which can then be applied to a switch. An overview of an OpenFlow set up is displayed in Figure 2, where the southbound interfaces are marked with a red color.

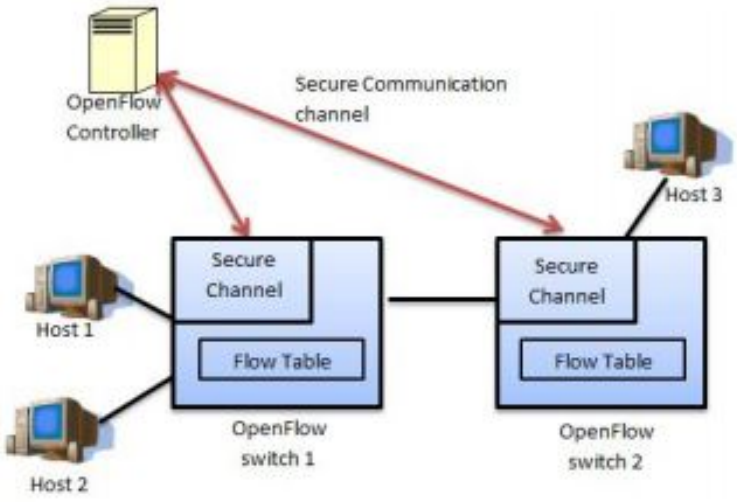

Figure 2: OpenFlow architecture. [16]

3.3 Network Function Virtualization

Network Function Virtualization (NFV) refers to the strategy of virtualizing network functions moving from separate proprietary hardware to software running on virtual servers using standard hardware. Using proprietary hardware and upgrading them is costly. While NFV reduces the costs of the hardware needed by decoupling functions from dedicated hardware and moving the functions to a virtual server. These functions could for example be a firewall, encryption, DHCP, NAT or virtual switches. Instead of installing expensive proprietary hardware, inexpensive switches, storage and servers could be purchased to run Virtual Machines (VMs) to perform network functions[12]. The difference between traditional network and NFV approach can be illustrated in Figure 3.

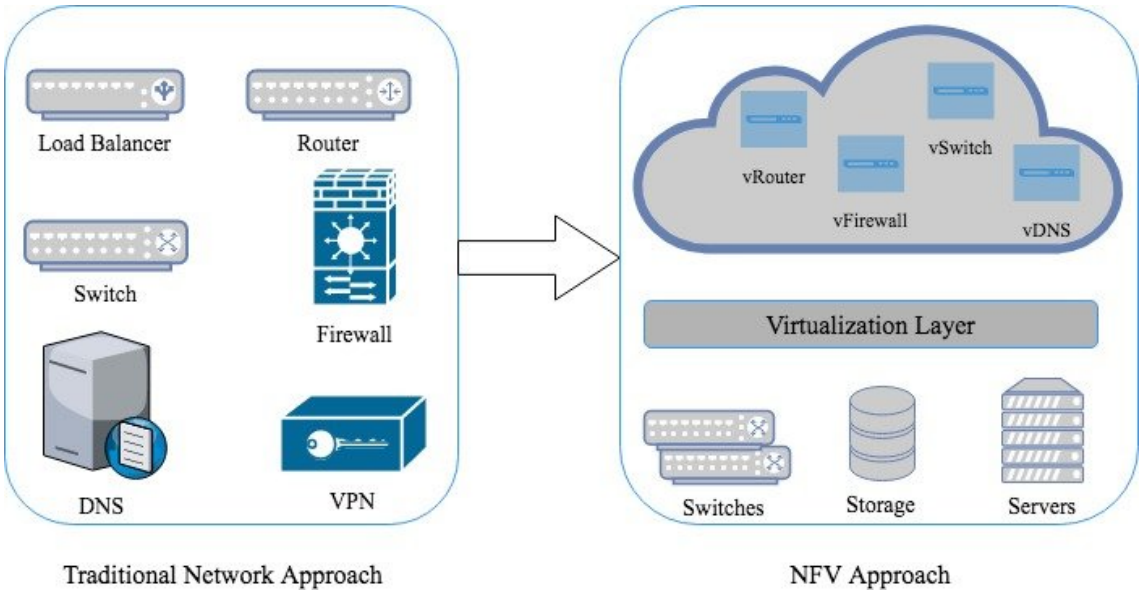

Figure 3: Difference between traditional network and NFV approach. [4]

3.4 Open vSwitch

Open vSwitch (OVS) is an open source software application that allows one to deploy and manage virtual switches on all major hypervisor platforms [20]. One of the main benefits of using Open vSwitch over other virtual switches is its capability to work with OpenFlow. This allows one to completely reprogram the working of the switch allowing it to be used for many different purposes.

The packet forwarding in Open vSwitch is done using two main components. These two components are the ovs-vswitchd and kernel datapath as is displayed in Figure 4. The ovs-vswitchd component is used to communicate with the OpenFlow controller (Faucet in this case), while the kernel datapath is used to handle the packets. If a packet was to enter the switch for the first time, the kernel datapath component would not know what to do with it. In that case, the kernel datapath would ask the ovs-vswitchd for instructions. The ovs-vswitchd should then provide instructions back to the kernel datapath component and ask it to cache the instructions. The instructions provided by the ovs-vswitchd are derived from the logic it has received from the SDN controller.

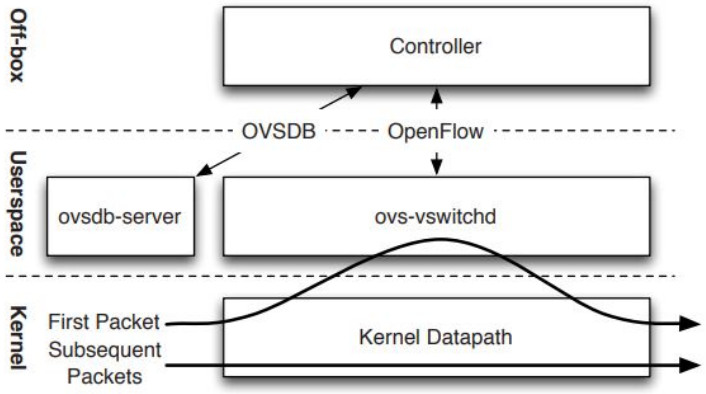

Figure 4: Open vSwitch architecture. [20]

4 Methodology

4.1 Approach

To answer the research questions, an emulation of an ICS environment should be created. Due to the fact that setting up a real ICS environment requires specific hardware, a virtual environment will be created for this research. All experiments will be run on an Ubuntu server equipped with a Xen hypervisor [23]. On the server two virtual machines (VMs) will be created. Each VM will be equipped with 2 virtual CPUs, 4 Gigabytes of RAM. On one of the VMs, OpenPLC will be installed to simulate a PLC. OpenPLC is an open source software application that can be run on OpenPLC compatible hardware [3]. Since it is also used in production ICS environments, it should resemble realistic behaviour and load on the network by a PLC. On the other VM, Faucet will be installed as an SDN controller. Faucet is an open source software application that can be used to serve as an SDN controller [5]. It has support for Open vSwitch built-in and is built to be used in High Availability (HA) environments, like ICS environments. These two VMs will then be connected together using Open vSwitch switches, like is displayed in Figure 5.

After the topology is created, three different NFV scenarios will be implemented in Python to detect unreported failures and/or to mitigate them. The first scenario will be to try to re-route traffic in case of a hardware failure. In this scenario, one of the switches will be destroyed and the network connection should automatically be re-established by re-routing the traffic to the other switch.

In the second scenario, one of the switches will be destroyed but now instead of re-routing the traffic to the other switch, the NFV should redeploy the switch which should re-establish the connection between the two VMs. This will be achieved by extending the *datapath_disconnect* in the *valve.py* file of the Faucet source code. The code checks which switch goes down and redeploys that switch. The redeployment command is designed to log into the Dom0 machine over a Secure SHell connection (SSH). Once logged in, the command executes an Open vSwitch command that will recreate the switch that failed. The final implementation of this function is displayed in Listing 2 in Section 9.

In the last scenario, if one of the interfaces on one of the switches stops transmitting, the NFV should detect this and recreate the interface on the switch, which should re-establish the connection between the two VMs. To implement this, knowledge about what switches

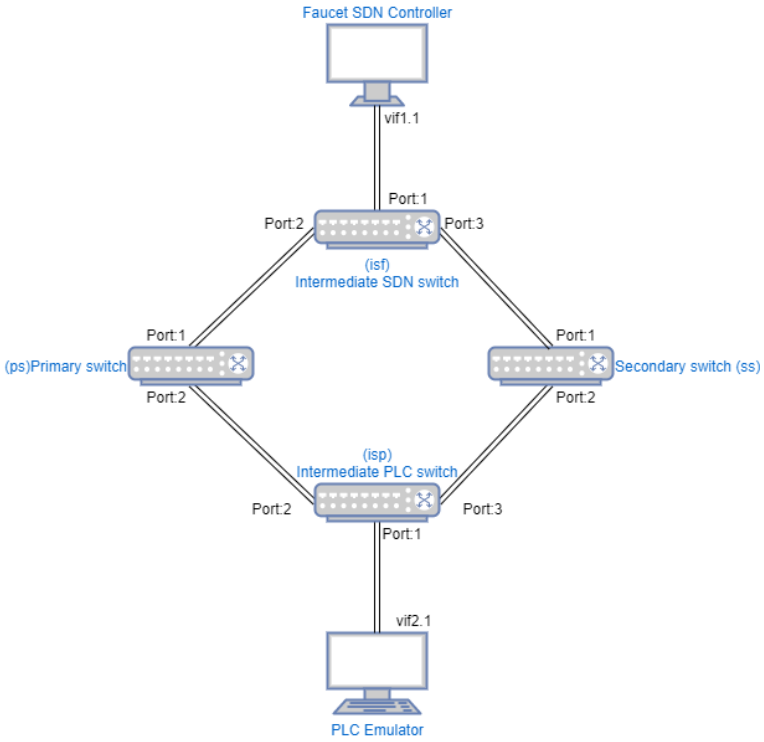

Figure 5: Virtualized ICS environment used for the experiments.

are present is needed first. This can be acquired by listing the current switches with the following command *ovs-vsctl list-br*. Then, since now we have the knowledge of our network, we are able to perform checks on each switch and its connected ports. In order to determine that there is a faulty port we will be looking into the transmit values that a specific port has with the help of the command *ovs-ofctl dump-ports <switch-name >*. By running this command we are able to observe how much traffic is transmitted but as well as how much traffic is received. The output of this command can be found in Listing 1. Whenever the transmit value is not increasing (TX value), there is no more traffic being transmitted through that port. Therefore we can say with confidence that the port is faulty. Since in a network with a larger amount of components (e.g. 104 switches and 202 hosts) it takes time to evaluate the status of all switches and its ports, an interval of 2 seconds is required in order to prevent the computer from overtaking itself. Choosing an interval too small showed to cause errors in the OVS daemon. The final implementation of this function is illustrated in Listing 2 in Section 9.

```
OFPST_PORT reply (xid=0x2): 3 ports
  port LOCAL: rx pkts=0, bytes=0, drop=0, errs=0, frame=0, over=0,
           crc=0, tx pkts=0, bytes=0, drop=0, errs=0, coll=0
  port  "vif1.1": rx pkts=179, bytes=10072, drop=0, errs=0, frame=0,
           over=0, crc=0, tx pkts=104045, bytes=7283406, drop=0,
           errs=0, coll=0
  port  "vif2.1": rx pkts=179, bytes=10072, drop=0, errs=0, frame=0,
           over=0, crc=0, tx pkts=104043, bytes=6242906, drop=0,
           errs=0, coll=0
```

Listing 1: Switch metrics regarding its connected ports.

To benchmark the impact on the downtime, we will take ten measurements for each scenario. In this research we define downtime as a machine that is not able to transceive data over the

network. Each measurement will be taken by starting connectivity check (ping) from the Faucet VM to the OpenPLC VM, configuring it to send packets every 10 ms. After starting a connectivity check, the switch or interface will be removed or disabled and the amount of dropped packets was calculated. Next, the experiments will be repeated with 54 switches and 102 hosts and 104 switches and 202 hosts.

Finally, this research will look into the advantages and limitations of each scenario. A comparison between the scenarios will be made.

4.2 Scope

The general scope of this project is to enhance the availability of the network of an ICS environment. In order to achieve that, we have to detect and identify what the problem is and therefore, by a dynamic decision, an NFV will be executed. The network should then recover according to the scenario as explained in Subsection 4.1.
Thus, the scope of our research was limited to finding appropriate and suitable solutions for an ICS environment using SDN and network functions in order to enhance availability. More precisely, redeploying a switch, recreating a faulty port or re-routing traffic through an alternative path that is available. This can be achieved by monitoring the network's hardware health and status and, if necessary, the NFV will take action according to the event that happened. This also limits the necessity of human intervention, as the remediation will be automatic.

5 Results

5.1 Experiment 1 - Re-routing scenario

The results of the experiments of the first scenario are listed in Tables 1, 2 and 3. For the first experiment, in which four switches and two hosts were present, the mean downtime observed was 5576 ms with a standard deviation of 1356 ms. Looking at the second experiment, where 54 switches and 102 hosts were present, a mean downtime of 7217 ms could be observed with a standard deviation of 3548 ms. Finally, for our last experiment, where 104 switches and 202 hosts were present, a mean downtime of 10050 ms could be observed with a standard deviation of 2922 ms. The average results of all three experiments are displayed in Figure 6.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	732	305	427	4270
2	683	242	441	4410
3	898	247	651	6510
4	679	160	519	5190
5	808	247	561	5610
6	788	239	549	5490
7	646	169	477	4770
8	808	166	642	6420
9	1168	299	869	8690
10	636	196	440	4400

Table 1: Experiment 1 - Re-routing scenario - four switches - two hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	690	308	382	3820
2	1312	433	879	8790
3	1824	201	1623	16230
4	977	260	717	7170
5	1035	230	805	8050
6	774	175	599	5990
7	898	222	676	6760
8	654	214	440	4400
9	881	241	640	6400
10	622	166	456	4560

Table 2: Experiment 2 - Re-routing scenario - 54 switches - 102 hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	770	292	478	4780
2	1364	342	1022	10220
3	1347	186	1161	11610
4	1480	147	1333	13330
5	1434	299	1135	11350
6	1449	241	1208	12080
7	1346	220	1126	11260
8	1056	163	893	8930
9	1325	144	1181	11810
10	646	133	513	5130

Table 3: Experiment 3 - Re-routing scenario - 104 switches - 202 hosts.

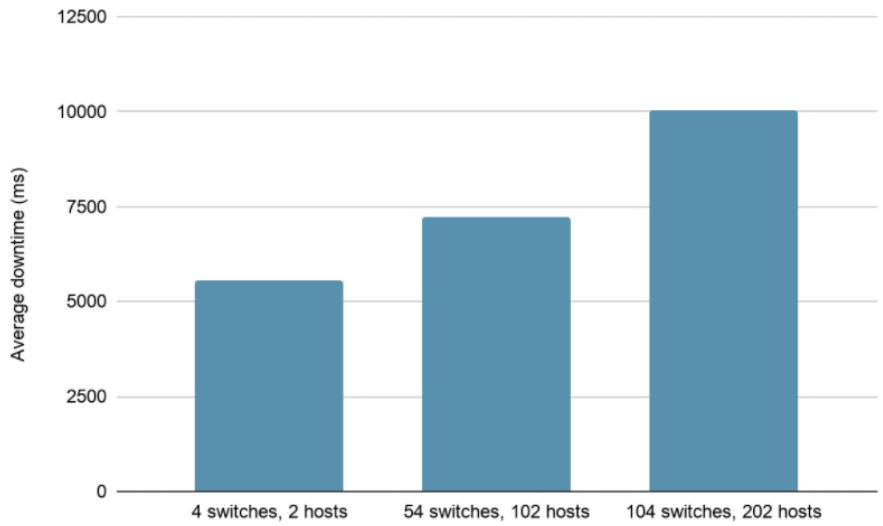

Figure 6: Average amount of downtime in milliseconds in case of recovering the network connection by re-routing traffic.

5.2 Experiment 2 - Switch redeployment scenario

After the implementation of the redeployment code was done in the Faucet source code, the performance measurements were taken in the three scenarios as is described in Section 4. The results of these measurements are displayed are in Tables 4, 5 and 6. The averages of the results of the experiments are 48 ms, 57 ms and 130 ms with a standard deviation of 6, 9, 9 ms respectively. These averages are visualized in Figure 7.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	202	197	5	50
2	197	192	5	50
3	246	241	5	50
4	235	230	5	50
5	271	267	4	40
6	239	235	4	40
7	255	251	4	40
8	231	226	5	50
9	254	248	6	60
10	440	435	5	50

Table 4: Experiment 1 - Switch redeployment scenario - four switches - two hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	377	371	6	60
2	235	227	8	80
3	232	225	7	70
4	254	247	7	70
5	254	246	8	80
6	241	233	8	80
7	294	287	7	70
8	242	234	8	80
9	334	327	7	70
10	209	200	9	90

Table 5: Experiment 2 - Switch redeployment scenario - 54 switches - 102 hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	263	251	12	120
2	196	183	13	130
3	314	300	14	140
4	285	271	14	140
5	219	206	13	130
6	276	263	13	130
7	212	199	13	130
8	211	197	14	140
9	224	213	11	110
10	215	202	13	130

Table 6: Experiment 3 - Switch redeployment scenario - 104 switches - 202 hosts.

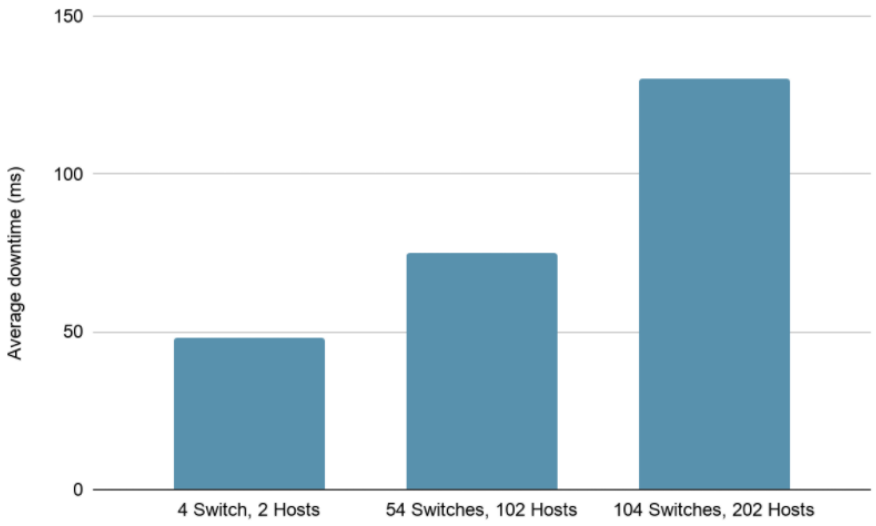

Figure 7: Average amount of downtime in milliseconds in case of recovering the network connection by redeploying the failed switch.

5.3 Experiment 3 - Recreation of a faulty port scenario

The performance measurements were taken in three scenarios as mentioned before in the previous experiments. The results of these measurements are displayed in Tables 7, 8 and 9. The averages of the results in the tables are 507 ms, 608 ms, 1254 ms with a standard deviation of 288 ms, 329 ms, 649 ms respectively. These averages are displayed in Figure 8.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	227	203	24	240
2	206	151	55	550
3	228	143	85	850
4	167	154	13	130
5	192	121	71	710
6	145	123	22	220
7	147	116	31	310
8	192	125	67	670
9	170	128	42	420
10	241	144	97	970

Table 7: Experiment 1 - Recreation of a faulty port scenario - four switches - two hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	290	217	73	730
2	349	304	45	450
3	222	210	12	120
4	249	160	89	890
5	244	173	71	710
6	258	155	103	1030
7	245	230	15	150
8	267	174	93	930
9	243	167	76	760
10	207	176	31	310

Table 8: Experiment 2 - Recreation of a faulty port scenario - 54 switches - 102 hosts.

Experiment	Packets sent	Packets received	Packets lost	Downtime (ms)
1	238	176	62	620
2	291	160	131	1310
3	245	192	53	530
4	266	199	67	670
5	283	157	126	1260
6	287	204	83	830
7	298	158	140	1400
8	310	146	164	1640
9	301	142	159	1590
10	422	153	269	2690

Table 9: Experiment 3 - Recreation of a faulty port scenario - 104 switches - 202 hosts.

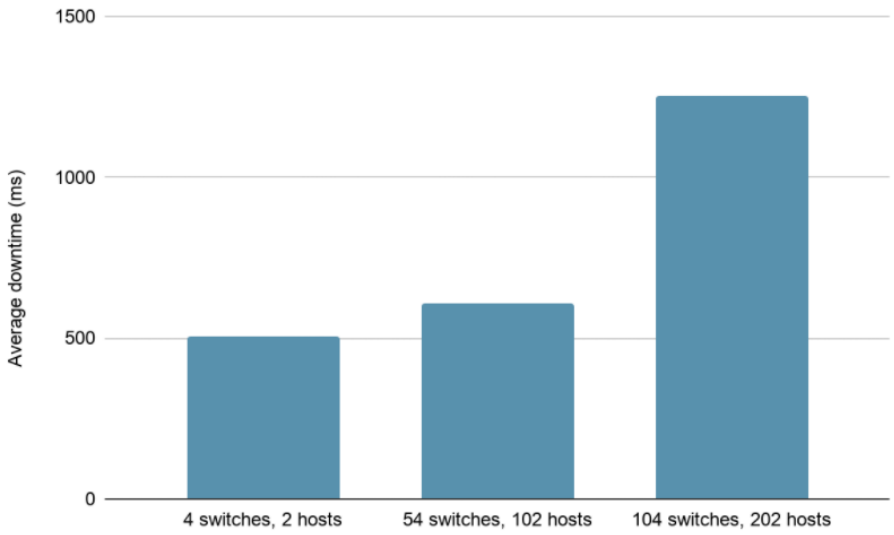

Figure 8: Experiment 1 - Switch redeployment scenario - four switches - two hosts.

5.4 Average downtime comparison

When comparing these results, as is displayed in Figure 9, we can observe that redeploying a switch is faster than recreating a faulty port. However, recovering a faulty port has a lower amount of downtime in comparison to the scenario of re-routing traffic. Moreover, the amount of downtime a PLC will have grows exponentially while the amount of switches and hosts grows linearly. This is true for all three scenarios performed in this research.

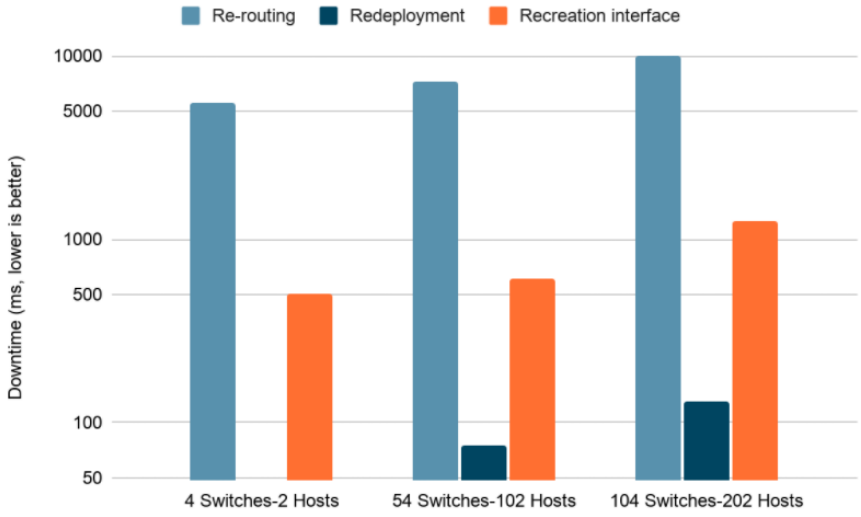

Figure 9: Comparison of the average downtime of all scenarios per experiment.

6 Discussion

6.1 Network Functions

Through the experiments we were able to observe that the least amount of downtime (48ms, 75ms, 130ms) occurs when a redeployment of a new switch is performed. However, there is a limitation, one cannot replace actual hardware when it is faulty using these functions because this is only feasible when virtual switches are used. The same can be expected for recreating a port on a virtual switch. On the other hand, our first experiment, which was about re-routing traffic to an alternative path that is available from a source to a destination, could be implemented on actual hardware. However, this scenario showed to be the slowest scenario (5576ms, 7217ms, 10050ms) out of all scenarios proposed in this research. Moreover, regarding our third scenario (average downtime: 507ms, 608ms, 1254ms) we had to set a delay in between every check since evaluating 104 switches in order to check whenever traffic is transmitted or not takes time. The time interval we set for this interval is 2 seconds. This reduces the performance of our scenario since a 2 seconds delay will be present between every check. Having more hardware present on the network will require even more time to evaluate and therefore a modification on the code should be made. Furthermore, the performance of the scenarios may not be sufficient for all ICS environments since there is still some downtime present. If a downtime of 48-10050 ms can be allowed, depending on the amount of hardware, the scenarios might still be sufficient. According to the United States National Institute of Standards and Technology (NIST), every organization that runs an ICS environment should define what loss of communications means (i.e. downtime), since

the downtime allowed could be different [25]. This means that our solution would have to be evaluated before implementing it in an ICS environment. Some ICSs would prefer to trade simplicity for the speed of recovery in case of a hardware failure, availability and also the ability of having more machines running while using less hardware.

6.2 Virtualization Architecture

During this research, as we mentioned in Section 4, we used Xen as our hypervisor to deploy two virtual machines with two virtual CPUs. However, using a different hypervisor could provide different results. Since we used a type two hypervisor which requires a host operating system (OS), more delays are likely to occur due to the fact that all instructions have to pass through the host OS (Dom0). While using a type one hypervisor, the guest operating systems will be able to communicate directly to the hardware and therefore is likely to result in better performance [18]. Furthermore, using virtual machines with more resources could also lead to better results since more computation power will be available to run the software on. In general testing our experiments on a different virtualization architecture (container-based, library OS on a hypervisor) could improve our results. Moreover, this research could be applied to a hardware infrastructure by having backup hardware available which could be enabled when a network hardware component would fail. However, using a virtualized infrastructure multiple virtual hardware components could be run on a single hardware component. This would also reduce the cost of hardware significantly, since this solution would require less hardware.

6.3 SDN Controller

The SDN controller we used is Faucet. However, Faucet is written in Python which is considered as a high-level programming language. An alternative SDN controller which is written in a lower-level programming language could lead to different results. Since Python code is interpreted by the Python runtime environment and not precompiled, the execution of the code takes more time [22]. For example looking into Floodlight, which is written in Java, Nox which is written in C++ or Trema which is written in Ruby and C could shed a light on the differences [14].

6.4 Metric detecting downtime

We used ICMP protocol for measurements, because it is bidirectional. The protocols used in an ICS environment are oftentimes based on the TCP protocol, which is also bidirectional (e.g. Modbus, DNP3, rs-232, rs-485, etc.) and thus will lead to a realistic scenario [9]. In the TCP and ICMP protocol, packets that are received by one side are acknowledged to the sending side. Since reliability is very important in ICS environments, it is crucial that instructions that are being send are acknowledged. Moreover, the ping command gives us the ability to set a shorter interval of how often packets should be transmitted. By doing so, a shorter interval did help us to perform more accurate experiments and therefore to get more accurate results.

6.5 Network Topology

Since some of the PLCs in an ICS environment do not support any modifications, modifying a PLC is often not a viable solution. Sometimes, this is because a PLC is dated, other times modifying all PLCs causes instability and a manual reconfiguration will have to be performed [24]. In order to avoid that, an intermediate switch should be present in order to listen to OpenFlow protocol and forward traffic to an alternative path in case of a failure on the primary path. However, there is a single point of failure in this situation as is illustrated in Figure 5. That being said, our scenario to redeploy a switch in case of a failure

could be implemented, which would re-establish the connection in approximately 48-130 ms, depending on the amount of hosts and switches that are present on the network.

7 Conclusion

In this paper, three different scenarios were implemented to combine SDN technology with NFV to improve availability in ICS environments have been researched. To do this, three different NFV functions were implemented. The experiments consisted of: re-routing traffic, redeploying a switch and redeploying a specific interface on the switch if connectivity with the switch was lost. These implementations were then applied by altering the Faucet SDN controller's source code. To allow the Faucet SDN controller to redeploy switches or interfaces of a switch, a connection between the controller and the Open vSwitch (OVS) daemon had to be created. This showed to be possible by implementing code in the Faucet controller that would execute commands to provide the Open vSwitch daemon with the right instructions. By monitoring the health of the switches and its ports periodically, the NFV was able to detect when a switch or port became unavailable. In the scenario where the data would be rerouted, the switches received an instruction to reroute the traffic to a different switch to restore the connectivity. In the scenario where a switch would be redeployed as a whole, the OVS daemon would receive the command to first delete and then create the switch again. In the last scenario where a single interface on a switch would be replaced, the OVS daemon would be instructed to first remove the faulty interface and then recreate it. This way backup network equipment could be provisioned to enhance availability. This answers our first sub-question: *How can SDN combined with NFV provision back-up network equipment to maintain availability during a network failure?*.

As for the performance of all three scenarios, a clear distinction can be made. While all scenarios were able to restore connectivity, variation in the amount of downtime between failure and recovery of the connection was visible. The re-routing of the traffic to a different switch took the longest, taking 5576 ms on average to restore connectivity in a network of four switches and two hosts. This is probably due to the controller having to recalculate the route the data needs to travel over, which could take some time. The recreation of an interface on a switch turned out to be faster, taking 507 ms in the same situation. However, the scenario that tried to redeploy a switch completely showed to be the fastest: only 48 ms were required to restore connectivity between the two hosts on average. In all cases, the connection between the Faucet SDN controller, the switches and their ports was re-established. Meaning that updates from the Faucet controller would still be obeyed by the network devices. This answers our second sub-question: *What are the consequences of provisioning back-up network equipment in an ICS environment for the manageability and connectivity of the network and its connected PLCs?*.

While these scenarios did reduce the average downtime in different situations, the scenarios also have their limitations. Because the NFV functions will have to monitor all switches and their ports periodically, CPU power and network bandwidth is required to provide the NFV with this information and to allow for evaluation of the data. Moreover, because all switches need to be evaluated, some time is required to complete all evaluations. In this research at least two seconds was required to let the NFV evaluate the state of 104 switches and 202 hosts. It is likely that in case the amount of switches and hosts would increase further, the amount of time it would take to evaluate all these devices would increase as well. This means that it will take some time before the NFV will check on a specific switch again, introducing a delay which results in downtime. Furthermore, because these scenarios used software functions to provision backup hardware, it will probably be less effective when implemented in a hardware environment. Since hardware most of the times cannot be redeployed using a software call, manual intervention would still be required to replace the

hardware part. This answers our third sub-question: *What are the limitations of using SDN combined with NFV in an ICS environment regarding the availability of the connected PLCs?*.

After evaluating these results, we can answer our main research question: *How could Software Defined Networking combined with Network Function Virtualization enhance availability in an Industrial Control System in case of a network hardware failure?*. One may conclude that SDN combined with NFV could enhance the availability in an ICS environment in case of a network hardware failure. Due to dynamic decision making, the NFV scenarios showed to be able to deploy backup virtualized hardware, recreate ports on a switch and reroute traffic in case of a failure, downtime in an ICS environment could be enhanced significantly. Reducing the risk of incidents and reducing the amount of required human intervention.

8 Future Work

Since this research has primarily focused on experiments in a virtualized ICS environment, the future research could focus on what significant performance differences there could be between a virtual ICS environment and a hardware ICS environment.

Moreover, in this research only one specific SDN controller has been used. A future research could validate if a different SDN controller would make a difference in the amount of downtime of each scenario that was tried. A different SDN controller, especially one that is written in a different programming language than Python, could lead to different performance of the NFV and therefore different results. One could for example look into Floodlight, which is written in Java [14]. Another SDN controller that could be used for comparison is Nox. Since Nox is written in C++, which is considered a lower level language than Python, it should have the potential to be more efficient than the Python based Faucet [10].

Finally, choosing a different programming language for our NFV could lead to the minimization of the interval we had for checking (2 seconds) since more efficient code could do checks more frequently. Moreover, looking into network size limitations would be interesting since there was an exponential increase regarding the downtime when more hardware was present on the network, therefore a research could be conducted to find out what are the limitations of our NFV code.

References

[1] Khandakar Ahmed et al. "Software defined networks in industrial automation". In: *Journal of Sensor and Actuator Networks* 7.3 (2018), p. 33.

[2] Otis Alexander, Misha Belisle, and Jacob Steele. *MITRE ATT&CK® for industrial control systems: Design and philosophy*. 2020.

[3] Thiago Rodrigues Alves et al. "OpenPLC: An open source alternative to automation". In: *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. IEEE. 2014, pp. 585–589.

[4] Ahmed Alwakeel, Abdulrahman Alnaim, and Eduardo Fernández. "Toward a Reference Architecture for NFV". In: May 2019, pp. 1–6. DOI: 10.1109/CAIS.2019.8769449.

[5] Josh Bailey and Stephen Stuart. "Faucet: Deploying SDN in the enterprise". In: *Communications of the ACM* 60.1 (2016), pp. 45–49.

[6] Adrian R Chavez. "Parametrization and Effectiveness of Moving Target Defense Security Protections for Industrial Control Systems". In: (2017).

[7] cisa. *ICS-CERT Alerts*. 2020. URL: https://us-cert.cisa.gov/ics/alerts?page=0.

[8] Rui Miguel da Conceição Queiroz. *Integration of SDN technologies in SCADA Industrial Control Networks*. 2017. URL: https://estudogeral.sib.uc.pt/bitstream/10316/83367/1/Relat%c3%b3rio%20de%20Estagio%20-%20versao%20FINAL_pos%20correcoes_v3.pdf.

[9] Z. Drias, A. Serhrouchni, and O. Vogel. "Taxonomy of attacks on industrial control protocols". In: *2015 International Conference on Protocol Engineering (ICPE) and International Conference on New Technologies of Distributed Systems (NTDS)*. 2015, pp. 1–6. DOI: 10.1109/NOTERE.2015.7293513.

[10] Daniel Frampton et al. "Demystifying magic: high-level low-level programming". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*. 2009, pp. 81–90.

[11] Bo Han et al. "Network function virtualization: Challenges and opportunities for innovations". In: *IEEE Communications Magazine* 53.2 (2015), pp. 90–97.

[12] Nerea Toledo Juanjo Unzilla Jon Matias Jokin Garay and Eduardo Jacob. "Toward an SDN-Enabled NFV Architecture". In: 2015, pp. 187–193.

[13] György Kálmán. "Prospects of Software-Defined Networking in Industrial Operations,"". In: *International Journal on Advances in Security* 9.3 (2016).

[14] Zuhran Khan Khattak, Muhammad Awais, and Adnan Iqbal. "Performance evaluation of OpenDaylight SDN controller". In: *2014 20th IEEE international conference on parallel and distributed systems (ICPADS)*. IEEE. 2014, pp. 671–676.

[15] Dong Seong Kim, Fumio Machida, and Kishor S Trivedi. "Availability modeling and analysis of a virtualized system". In: *2009 15th IEEE Pacific Rim International Symposium on Dependable Computing*. IEEE. 2009, pp. 365–371.

[16] Adrian Lara, Anisha Kolasani, and Byrav Ramamurthy. "Network innovation using openflow: A survey". In: *IEEE communications surveys & tutorials* 16.1 (2013), pp. 493–512.

[17] Pavlos Lontorfos. "Securely accessing remote sensors in critical infrastructures." In: (2020).

[18] R. Morabito, J. Kjällman, and M. Komu. "Hypervisors vs. Lightweight Virtualization: A Performance Comparison". In: *2015 IEEE International Conference on Cloud Engineering*. 2015, pp. 386–393. DOI: 10.1109/IC2E.2015.74.

[19] Andrés Felipe Murillo Piedrahita et al. "Securing virtual industrial control systems using SDN/NFV platforms". PhD thesis. Uniandes, 2019.

[20] Ben Pfaff et al. "The design and implementation of open vswitch". In: *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*. 2015, pp. 117–130.

[21] Andrés F Murillo Piedrahita et al. "Leveraging software-defined networking for incident response in industrial control systems". In: *IEEE Software* 35.1 (2017), pp. 44–50.

[22] Programiz. *Interpreter Vs Compiler : Differences Between Interpreter and Compiler*. 2021. URL: https://www.programiz.com/article/difference-compiler-interpreter.

[23] The Linux Foundation Project. *Xen Project*. 2020. URL: https://xenproject.org/.

[24] Carl D Schuett. "Programmable logic controller modification attacks for use in detection analysis". In: (2014).

[25] Keith Stouffer, Joe Falco, and Karen Scarfone. "Guide to industrial control systems (ICS) security". In: *NIST special publication* 800.82 (2011), pp. 16–16.

[26] Wikipedia. *Software-defined networking*. 2020. URL: https://en.wikipedia.org/wiki/Software-defined_networking.

[27] Luying Zhou and Huaqun Guo. "Applying NFV/SDN in mitigating DDoS attacks". In: *TENCON 2017-2017 IEEE Region 10 Conference*. IEEE. 2017, pp. 2061–2066.

9 Appendix

```
    def datapath_disconnect(self, now):
        """Handle Ryu datapath disconnection event."""
        self.logger.warning('[RP2] SWITCH IS DOWN!!!!!')
        self.logger.warning('this switch with number '+ str(self.dp.
            dp_id)+' is down.')
        self.logger.warning(str(self.dp))
        self.logger.warning('datapath down')
        self.notify(
            {'DP_CHANGE': {
                'reason': 'disconnect'}})
        self.dp.dyn_running = False
        self._inc_var('of_dp_disconnections')
        self._reset_dp_status()
        self.ports_delete(self.dp.ports.keys(), now=now)

        if str(self.dp) == 'ps':
            os.system('sudo ssh root@145.100.104.125 "sudo ovs-vsctl
                add-br ps -- set bridge ps other-config:datapath-id
                =0000000000000001 -- set bridge ps other-config:
                disable-in-band=true -- set bridge ps fail_mode=secure
                 -- set-controller ps tcp:145.100.111.130:6653 -- add-
                port rs vif1.1 -- set interface vif1.1 ofport_request
                =1 -- add-port rs vif2.1 -- set interface vif2.1
                ofport_request=2"')
        else:
            os.system('sudo ssh root@145.100.104.125 "sudo ovs-vsctl
                add-br ss -- set bridge ss other-config:datapath-id
                =0000000000000002 -- set bridge ss other-config:
                disable-in-band=true -- set bridge ss fail_mode=secure
                 -- set-controller ss tcp:145.100.111.130:6653 -- add-
                port ss vif1.2 -- set interface vif1.2 ofport_request
                =1 -- add-port ss vif2.2 -- set interface vif2.2
                ofport_request=2"')
```

Listing 2: NFV function extension to recreate switches

```
import os
import threading
import time
import subprocess

prevEvals = {}
prevInterfaces = {}

def main():
  evaluateSwitches()

def evaluateSwitches():
  threading.Timer(2, evaluateSwitches).start()
  switches = getSwitches()

  for switch in switches:
    if switch in prevEvals.keys():

      if switch in prevInterfaces.keys():
        interfaces = getSwitchInterfaces( switch )
```

```
          interfaceDiff = list( set( prevInterfaces[ switch ] ) -
              set( interfaces ) )

          for interface in interfaceDiff:
            os.system('sudo ovs-vsctl -- add-port ' + switch + ' ' +
                 interface + ' -- set interface ' + interface + '
                ofport_request=' + interface.split('st')[1])

          txValues = getTxFromSwitch( switch )

          interfaces = getSwitchInterfaces( switch )
          interfaceDiff = list( set( prevInterfaces[ switch ] ) -
              set( interfaces ) )

          if len( interfaceDiff ) > 0:
            break

          for interface in interfaces:
            if ( interface in prevEvals[ switch ] and txValues[
                interface ] == prevEvals[ switch ][ interface ] ):
              os.system('ovs-vsctl del-port ' + switch + ' ' +
                  interfaces[i])
              os.system('sudo ovs-vsctl -- add-port ' + switch + ' '
                   + interfaces[i] + ' -- set interface ' +
                  interfaces[i] + ' ofport_request=' + str(i + 1))

              break

      if switch not in prevInterfaces.keys():
        prevInterfaces[ switch ] = getSwitchInterfaces( switch )

      prevEvals[ switch ] = getTxFromSwitch( switch )

  def getSwitchInterfaces( switch ):
    switchInfo = os.popen('ovs-vsctl list-ports ' + switch ).read()

    return switchInfo.split()

  def getSwitches():
    switches = os.popen('ovs-vsctl list-br').read()

    return switches.split()

  def getTxFromSwitch( switchId ):
    report = os.popen('sudo ovs-ofctl dump-ports ' + switchId).read
        ()
    report = report.split('port')

    portReports = []
    txValues = {}

    for interfaceLine in report:
      if 'LOCAL' not in interfaceLine:
        portReports.append( interfaceLine )

    portReports = portReports[2:]
    switchInterfaceNames = prevInterfaces[ switchId ]
```

```
    for portReport in portReports:
      portReportTxString = portReport.split('tx pkts=')[1]
      portReportTxInt = int(portReportTxString.split(',')[0])
      interfaceId = str( switchInterfaceNames[ int( portReport.split
          ('tx pkts=')[0].strip(' ')[:1] ) - 1 ] )
      txValues[ interfaceId ] = portReportTxInt

    return txValues

  if __name__ == "__main__":
    main()
```

Listing 3: NFV function to evaluate interfaces