DEPARTAMENT OF COMPUTER SCIENCE

RESEARCH PROJECT:

INTRUSION DETECTION SYSTEMS IN SDN-BASED SELF-HEALING PMU NETWORKS

Spring Semester 2016

Written by:
Carlos De Las Muñecas *

Intrusion Detection Systems in SDN-based Self-Healing PMU Networks

Carlos De Las Muñecas, *CS Department, IIT*
*in collaboration with Adrian Tirados, *CS Department, IIT*

***Abstract*—Nowadays, Power grids are critical infrastructures on which everything else relies, and their correct behavior is of the highest priority. New smart devices are being deployed to be able to manage and control power grids more efficiently and avoid instability. However, the deployment of such smart devices like Phasor Measurement Units (PMU) and Phasor Data Concentrators (PDC), open new opportunities for cyber attackers to exploit network vulnerabilities. If a PDC is compromised, all data coming from PMUs to that PDC is lost, reducing network observability.**

Our approach to solve this problem is to develop an Intrusion detection System (IDS) in a Software-defined network (SDN). allowing the IDS system to detect compromised devices and use that information as an input for a self-healing SDN controller, which redirects the data of the PMUs to a new, uncompromised PDC, maintaining the maximum possible network observability at every moment.

During this research, we have successfully implemented Self-healing in an example network with an SDN controller based on Ryu controller. We have also assessed intrinsic vulnerabilities of Wide Area Management Systems (WAMS) and SCADA networks, and developed some rules for the Intrusion Detection system which specifically protect vulnerabilities of these networks.

The integration of the IDS and the SDN controller was also successful.
To achieve this goal, the first steps will be to implement an existing Self-healing SDN controller and assess intrinsic vulnerabilities of Wide Area Measurement Systems (WAMS) and SCADA networks. After that, we will integrate the Ryu controller with Snort, and create the Snort rules that are specific for SCADA or WAMS systems and protocols.

Keywords*—*Intrusion Detection System (IDS), Software-Defined Networking (SDN), Cybersecurity, Phasor Data Concentrator (PDC), Phasor Measurement Unit (PMU), Wide Area Measurement System (WAMS), SCADA.

I. Introduction

Nowadays, power grids are evolving and a new kind of smart power grid is being deployed in wide-area monitoring systems. In order to collect data from the grid, phasor measurement units (PMUs) are being installed. These devices are able to perform multiple measurements such as estimating the state of the grid, detecting and preventing power line outage etc.[1]
The measurements collected by PMUs are then delivered to a phasor data concentrator (PDC), which acts as an aggregator, receiving data from multiple PMUs and combining it before sending it to a higher level in the hierarchy, either a higher level PDC or directly the control center. The resulting system can be thought of as a hierarchical tree, with the control center at the top of the tree and the PMUs as the final elements or leaves.
The addition of intelligent devices to power grids has its own advantages and disadvantages:

- Advantages:
  - A smart grid will generally be much more efficient
  - Makes the grid easier to control
  - improves the speed of the response in case of failure.

- Disadvantages:
  - Every time you add smart, connected devices, this creates new vulnerabilities.
  - Adding smart devices such as PMUs creates the opportunity for cyber-attackers to interfere with the network.
  - Some recent studies show that PMUs and PDCs can be targeted by denial-of-service and man-in-the-middle attacks [2], [3].

A. Problem Statement

If a device is compromised or disconnected, part of the system will not be monitored, affecting the estimations and reducing the ability to detect anomalies in the power grid. This can lead to undetected failures in the system. This project will attempt to solve this problem by using an SDN-based self-healing network. In this context, the problem can be divided into two sub-problems:

1) The first issue is the detection of compromised devices. In order to use a self-healing mechanism, first the compromised device must be detected and isolated, to stop further infection if possible. To achieve this, we want to implement an intrusion detection system (IDS). This can be done by using Snort combined with the SDN controller. More

information on this can be found in Section III of this paper.

2) The second phase involves the self-healing of the network. In case the compromised device is a PMU, this is not too critical, because it is only one monitoring device that will fail to collect data. But if a PDC is compromised, all data sent to it by multiple PMUs will be lost. In order to collect that data, the traffic can be rerouted to another PDC. Specifically, an integer linear programming (ILP) model found in previous research is used to jointly minimize the overhead of re-configuring the communications network while considering the constraint of its hardware resources, such as the size of the forwarding table in switches. [4]

B. Similarities between WAMS and SCADA networks

For further research, it should be noted that we consider SCADA and WAMS networks to be equivalent, because the architecture behind them is really similar. For that reason, the solution given in this project may be developed for SCADA instead of WAMS depending on available resources and compatibility of the controllers and IDS mechanisms.

C. Network architecture

The network architecture of WAMSs can be seen in the image below. Several PMUs are logically connected to a single PDC,although they do not have to be directly connected, but there may be several switches between them. The PDC is also connected to either the control center or a next level PDC.
SCADA architecture has the same kind of architecture were multiple devices gather information and then send it to a higher level device, and they use a Master-slave communication system. In the SCADA architecture, it is normally PLCs that act as the analogue of the PMUs in WAMS. These PLCs act as slaves, and they are controlled my a Master which sends the control messages.

The remainder of this paper is organized as follows. Section II discusses relevant contributions by previous research. Section III discusses our research approach, including the different steps in our plan and examining the most challenging parts of this project. Section IV shows our results up to this moment, and section V gives our conclusions on the topic.

II. Research Approach

This section describes how we are focusing our research approach, and gives an introduction on each step. After reviewing the previous work in this field, we are ready to plan our line of research. This project will have several progressive phases, each one building on top of each other, apart from the first one which can be done in parallel with the rest of them. We believe this approach will have certain challenges that will be also addressed at the end of this section.

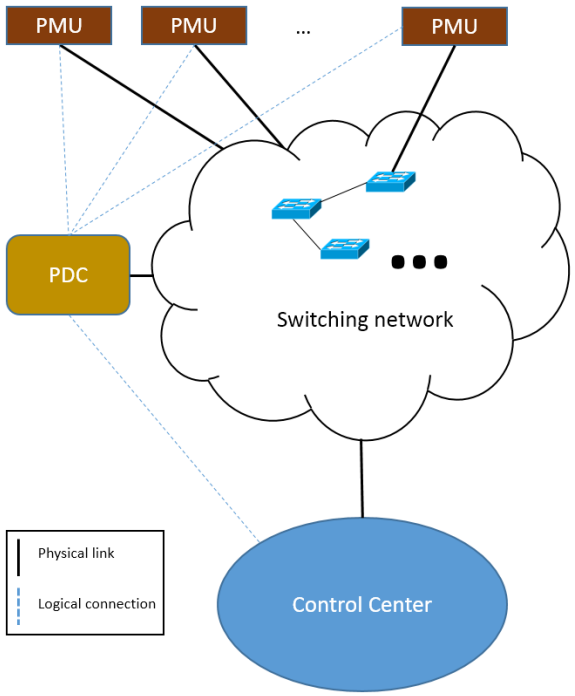

Fig. 1. WAMS network

A. Implementation of Self-Healing SDN Controller

This first phase addresses the need to implement the self-healing app for the Ryu controller proposed by [4]. At this moment, we have been able to replicate the solution via a hard-coded controller with limited functionality. We intend to extend this controller and add all the different functionalities and steps proposed in this previous work: recover observability first, and then maximize this observability by recovering all the remaining disconnected PMUs. In order to do so, we will solve the currently existing optimization problem for optimal reroute. Finally, we will evaluate our results in Mininet, a software platform for SDN, to demonstrate the self-healing mechanisms in a simulated network system.

B. SCADA/WAMS Protocol Study

SCADA Systems and Wide Area Measurement Systems (WAMS) share some specific similarities that can make our solution suitable for both architectures:

1) SCADA Systems monitor field instruments via Remote Terminal Units (RTU), while WAMS Systems make readings through Phasor Measurement Units (PMU).

2) Both architectures share data aggregators (Nodes in SCADA and Phasor Data Concentrators (PDC) in WAMS), which are used to combine the different measurements and send them to the next level

aggregator or control center.

However, SCADA and WAMS Systems differ in the amount and characteristics of data they monitor. On the one hand, SCADA usually has asynchronous information every few seconds. On the other hand, WAMS requires synchronization and reads every few milliseconds. Nevertheless, this fact does not suppose any mishaps in our work, since the preprocessor will potentially monitor the network the same way in both architectures.

This is the main reason why one of our first steps will be a deep study in SCADA and WAMS protocols, concretely Modbus and DNP3 for SCADA, and C37.118-2005 for WAMS. After we understand this protocols correctly, we will be able to complete the next phase.

C. Vulnerability Assessment

The next natural step to take is make a vulnerability assessment for SCADA and WAMS Systems. As we mentioned previously, there are already some works on this field that will set up the antecedents for this phase. We intend to provide a detailed study on these technologies, also focusing on the challenges that the addition of Software-Defined Networking can bring to the picture. The result of this vulnerability assessment will represent the starting point for our Intrusion Detection System rules' development.

D. Find a suitable Preprocessor

Preprocessors allow the functionality of Snort to be extended significantly. They can be used to either examine packets for suspicious activity or modify packets so that the detection engine can properly interpret them. Preprocessors are indispensable in discovering non-signature-based attacks, and they are responsible of normalizing traffic so that the detection engine can accurately match signatures. Another functionality worth mentioning is the ability to defeat attacks that attempt to evade Snort's detection engine by manipulating traffic patterns. Therefore, the preprocessor will be a key part of our system. It is important to conduct an extensive research in this area and try to identify the most suitable for our solution. Currently, the two main contributors to this field are Digital Bond (with a DNP3 preprocessor for Snort), and Cisco (with both Modbus and DNP3 solutions).

E. Ryu - Snort Integration

One of the most significant components of our novel approach is the combination of an SDN controller with an IDS. Due to previous works in this area, we have chosen Ryu as SDN controller and Snort as Intrusion Detection System. Ryu is a lightweight, component-based SDN framework that provides a well defined API that allows developers to create new network management and control applications. Snort is an open source IDS capable of real-time traffic analysis and packet logging.

Once the integration phase is completed, it will be tested with a simple controller application in a single-switch topology before starting the next development phase.

F. Snort Rules Development

This phase consists on developing specific Snort rules taking into account the expertise gained from the previous steps: the vulnerability assessment will allow us to specify certain rules to be checked, and the preprocessor will increase the functionality of the IDS running coordinately with the SDN controller. This work comprises the last design and development phase, prior to the final tests and evaluation of results.

G. Evaluation and Results

Finally, after all development phases have been completed, we will evaluate the performance of the proposed solution in terms of speed and accuracy. To do this, we will simulate an scenario on Mininet with the necessary SDN controller and IDS application. Additional results on scalability or impact in the network performance could be provided.

H. Challenges

It is worth mentioning a few challenges that we think we might encounter during this research project:

- Ryu-Snort integration: Even though there are already some works in this area, we believe that this step can be one of the most time consuming, both in the system work and testing work. Trying to integrate this two technologies in Power Grid networks is a novel approach and difficulties will arise.

- Vulnerability Assessment: SCADA and WAMS protocols have been less studied than the classic Internet protocols, which means that less architectural vulnerabilities have been discovered. This fact will put as in a more difficult position to determine a trustworthy, extensive list of vulnerabilities.

- IDS Rules: Due to our lack of expertise and previous work in this area, this can be another potential challenge in our research. In order to pave the way, we'll start training with classic protocol rules, so we can understand better how IDS systems work.

III. Related Works

There is some previous research about IDS in this kind of networks, especially in SCADA networks. In the paper *"Industrial Cybersecurity for power system and SCADA networks"*[5], the authors present an interesting high level analysis of the possible vulnerabilities and threats affecting a power plant environment, including information about commonly used devices and a discussion about intrinsic vulnerabilities of power plants.

There is more interesting research about security in SCADA networks in the paper *"Security Strategies for*

Scada Networks", where two different strategies are described to defend SCADA networks, and outlining some vulnerabilities that should be improve in order to secure this kind of architectures. The problem in these environments is usually that many of the communication protocols used in industrial environments (e.g. Modbus, DNP3 etc.) where not designed with security considerations in mind.
There are some interesting research papers about these protocols too, including one aiming to correct the vulnerabilities in DNP3.In the paper *"Distributed Network Protocol Security (DNPSec) security framework"* the authors present an extension to the protocol DNP3 which solves some of the vulnerabilities found in this protocol, adding Integrity,Authentication, non-repudiation etc.[7]

The use of classical ICT security countermeasures in order to protect the process network and the field network of a SCADA system have been proven inadequate [8], [9].
Traditional firewalls are usually unable to detect attack patterns specifically designed to exploit SCADA vulnerabilities, since most of this protocols are coded at application level and run over the TCP/IP communication stack.
Although it is true that Cisco started developing a prototype of Netfilter module with the ability to perform filtering analysis on single Modbus packets [10], this kind of mechanism can only detect malicious packets in the network, but is unable to identify complex attack where multiple licit commands are injected by an attacker in order to drive the system to unstable behaviors.

Most SCADA systems were originally designed for serial communication, when security was not a relevant factor for these control systems. These protocols (e.g. Modbus, DNP3), have then been ported to the application level to run over the classical TCP/IP protocol stack, introducing more complexity for the management of reliable delivery of control packets with strong real time constraints, and has introduced new vulnerabilities against cyber-attacks. Some of this vulnerabilities are:

- No integrity guaranteed: These protocols do not perform any kind of integrity checks on the control packets sent between master and slave, allowing easy alteration of the packets.
- No authentication: Anyone claiming to be the 'Master' can send commands to the slaves, because there are no authentication mechanisms.
- No anti-replay mechanisms

This makes it easy for attackers to perform several kinds of attacks such as [12]:

- Unauthorized execution of commands: Anyone who gains access to the network can send control messages to operate the slave devices
- DOS attack: Attackers can also forge meaningless Modbus/DNP3 packets,impersonating the Master, and consume all the resources.
- MITM-attacks: The fact that there is no integrity mechanism makes it easy for attackers to manipulate messages
- Replay attacks: There is no mechanism to repel this kind of attack either, so these systems are vulnerable to malicious replayed control messages.

All the previously discussed papers focus on the intrusion detection, but not on the self-healing part of our project.Self-healing in PMU networks can be achieved by exploiting the features of software-defined networking (SDN) to achieve resiliency against cyber-attacks [4]. Once a compromised device is detected, the switches in the network will be re-configure to isolate it and maintain connectivity of the PMUs with one PDC, ensuring the observability of the system.
This can be done with integer linear programming (ILP) model to minimize the overhead of the self-healing, with the constraints of power system observability, hardware resources, and network topology[4].

IV. Vulnerability Assessment

This section contains a thorough vulnerability assessment on WASM networks. This is a important part of our investigation, as it will provide the base on which the Snort Rules will be designed. The section is structured as follows: first, a brief introduction to the PMU protocol IEEE C37.118 allows the reader to understand the architecture and particularities of this protocol. Then, a set of vulnerabilities on this protocol are given, along with examples and possible outcomes from their exploits.

A. Background on IEEE C37.118

Although several standards and protocols support the WAMS infrastructure communications, we will focus on the latest version of the PMU/PDC protocol, IEEE C37.118 [13]. The main reason is that our protocol focuses on self healing compromised PMUs, therefore, we must analyze the protocol directly involved with this kind of devices.

IEEE C37.118 substituted on 1998 the previous IEEE 1344 synchrophasor protocol for PMUs. The main objective of IEEE C37.118 is improve the previous protocol as well as define clearly the format of data transmitted from PMU to PDC. In short, we can say that IEEE C37.118 is a protocol that defines synchrophasor data conventions, measurement accuracies and communication formats.

Thus, IEEE C37.118 introduces 4 different frames:

- Command: The contents of this frame if binary information that contains particular actions within the WAMS network. It is received by a PDC or PMU in order to perform a given action, and it is always sent

hierarchically from a PCD to a lower level PDC or PMU.

- Configuration: This binary frame contains the information and processing parameters of the PMU (e.g. phasor, frequency, number of analog values, conversion factors).

- Data: Containing only binary data as well, it provides information regarding the phasor data, frequency, estimates, etc.

- Header: The only header written on ASCII, thus, human readable. It contains information about the PMU, the source of the data, algorithms and other related information that can be useful for the administrator.

B. Vulnerabilities of IEEE C37.118

IEEE C37.118 does not support authentication, confidentiality or integrity on its own. If it is not combined with other security measures, it can lead to potential exploits that can compromise critical infrastructure networks. Some of the vulnerabilities that we have identified are:

- *Packet analysis*. TCP/IP packets sent from PMUs are clearly susceptible to eavesdropping and later analysis. Simple sniffing software could allow attackers to analyze traffic across the network if not encryption is used. For example, [15] uses Wireshark [16] to analyze packets in synchrophasor networks and found that they were being sent in plain text. One possible countermeasure is encrypting traffic end-to-end between PMU and PDC, with techniques like VPN tunneling via SSL/TLS [15]. However, we cannot forget that this solutions also have well-known vulnerabilities. These vulnerabilities will not be covered in this paper due to its limited extension, but there is a large amount of literature on the topic. Another possible countermeasures are analyzed on [22].

- *Packet injection*. In this possible attack, the attacker would send packets from a compromised device in the network to other PMUs or PDCs. This packets can contain instructions that can jeopardize the visibility of the system, or induce the PMUs to stop taking measurements. Furthermore, an attacker could be capable of injecting code or shell code to send malicious instructions to an existing database management system [17]. Finally, it would also be possible to insert wrong data in the network spoofing a PMU and hijack the readings. Previous works in this attack include [18], which injects false data into the system, and [19], in which researchers are able to spoof GPS time stamps of the measurements.

- *Man in the Middle Attacks*. This attack is one of the most extended, and PMU networks are not an exception. In the WAMS architecture, the attacker is positioned between PDC and PMU, making the PDC think that it is talking to the PMU and vice versa. This attack can result in compromised certificates. One clear countermeasure is provide authentication from client to server. Other works [20] suggest that with few extra PMUs, bad data detection and identification capability of a given system can be drastically improved.

- *Denial of Service*. Denial of Service (DoS) attacks are focused on exhausting the resources of their victims (e.g. CPU, bandwidth, memory) to prevent it from working correctly. This is critical in WAMS networks, since there is a great loss in visibility and control. There are multiple techniques [21] to achieve denial of service, here we will mention a few of them:
  - ICMP Smurf: The attacker spoofs the victim's IP address on the sender field and sends ICMP requests to multiple hosts that will reply to the victim, overwhelming its resources.
  - Fuzzing: This attack is based on the creation of random network packets with incorrect values in its fields, which can crash applications, services and soft rebooting due to the protocol mutations.
  - DDoS: The attacker is in control of a large amount of machines (bots), that can send ICMP packets to a victim with enough frequency to take it down.

  Feasible countermeasures can be the inclusion of filtering routers, disable IP broadcasting, and performing intrusion detection, which will be covered in the next section of the document.

- *Physical Attacks*. This last type of vulnerability lies within the physical aspect instead of the information security world, but it is also important noticing that these attacks can have catastrophic consequences. Unauthorized access to critical infrastructure can lead to the manipulation of the PMU or the injection of devices in otherwise safe networks.

V. MININET AND SDN CONTROLLER

In order to develop and experiment with the controller of a Software-Defined Network, there are two aspects to choose:

- The testing environment: There are several network simulation tools, although Mininet seems to be the simplest and best choice for an SDN network.

- The Controller: There are several open-source SDN controllers that can be used for this task, like NOX, POX, Floodlight, Ryu etc.

A. *mininet*

Mininet has been chosen as the proper environment to test the self-healing in a software defined network, because it it allows you to easily interact with your network using the Mininet client, customize it, or even deploy it on real hardware. Mininet is a very useful for development, learning and research. It is a reliable way to develop and experiment with OpenFlow and Software-Defined networks and is released under a permissive BSD Open Source license.

B. *SDN controller*

There are multiple options among open-source SDN controllers. Here we give a short list of their characteristics.

- NOX: It was the first highly popular OpenFlow controller. It was not heavily implemented primarily because NOX is programmed primarily in C++ and lacks good documentation. It supports OpenFlow 1.0 only.
- POX: It was NOX's successor, and has an easier development environment to work with and a reasonably well written API and documentation. It is written in Python, which typically shortens its experimental and developmental cycles. It supports OpenFlow 1.0 only.
- Ryu: It is similar to POX, but developed independently. Ryu provides software components with well defined API that make it simple to develop new network applications. It is also written in Python, and supports Openflow versions 1.0, 1.1, 1.2, 1.3, 1.4, 1.5 and the Nicira extensions.
- Floodlight: This one was a Fork from a previous controller 'Beacon'. It is written in Java, has a good documentation and a better performance than the other controller mentioned, but is not easy to develop and also lacks the support for later OpenFlow versions (just OpenFlow 1.0)

In the following table you can see a summary of their characteristics.

TABLE I. COMPARISON OF SDN CONTROLLERS

	NOX	POX	Ryu	Floodlight
Language	C++	Python	Python	JAVA
Distributed	No	No	Yes	Yes
Openflow version	1.0	1.0	1.0-1.5 + Nicira versions	1.0
Learning curve	Moderate	Easy	Moderate	Steep
Performance	Good	Medium	Medium	Good

In this research, the performance of the controller is not the highest priority, as the objective is to test self-healing algorithms in SDN networks and to integrate this with an Intrusion detection System. This is the reason Floodlight is not a good choice, as it is programmed in Java and has a steep learning curve.
Support for later versions is a desirable feature for future testing and development. For the previously stated reasons, we find Ryu to be the best fit for our project, as it provides a good API and good documentation, is written in Python, which makes it easy to develop, and supports multiple versions of OpenFlow.

VI. SNORT INTEGRATION

This section explains the SDN - Snort integration in our solution. It is divided in four subsections: first, we introduce a brief background on Snort and its rules; then, the final proposed architecture is presented; next, a set of rules is given as an example of this integration; and, finally, we discuss the benefits and drawbacks of this solution.

A. *Snort IDS*

SNORT is a popular open source Network Intrusion Detection System (NIDS) created by Martin Roesch in 1998. It is capable of performing real-time traffic analysis and packet logging on IP networks. There are three different operational modes available:

1) Sniffer: read network packets and display them on terminal.
2) Packet logger: log packets to the disk.
3) Network Intrusion Detection: analyze traffic against a rule set defined by the user.

The Network Intrusion Detection mode is the most suitable one for our solution, since we will run Snort as a NIDS monitoring several network interfaces. This will allow us to analyze traffic from PMUs to PDCs (and from PDCs to other top-level PDCs), and check it against a rule set specifically designed for PMU networks.
In Snort, rules are defined as strings of plain text composed of different parts (e.g. action, protocol, source and destination IP addresses, source and destination ports, and message and rule options for administrators). This flexibility is one of the main reasons why we chose Snort: we can easily develop rules for any protocol if we understand its architecture and vulnerabilities correctly.

B. *Architecture*

The architecture of this integration comprises one of the most important decisions of this solution. We have to analyze all the possible alternatives, both with advantages and disadvantages, in order to pick the most suitable one. We must analyze the architecture from two levels of perspective:

the first one, within the general PMU network, while the second one will address the device-level architecture.

When it comes to the design of the network architecture, there are two options to take into account:

- *Centralized architecture*. This architecture would perform a single Snort instance monitoring all the network interfaces at once in one of the central switches. To achieve this, we need to perform port mirroring from all the switches in the network to an interface that will be sniffed by the Snort IDS.
  The information collected by Snort will be sent to the controller via one of the two options that will be analyzed later.
  The main advantage of this solution is the fact that the maintenance and set up will be easier as we just need to take care of one Snort instance. However, a centralized architecture will have a worse scalability and possible bottlenecks if the traffic scales as well, as the amount of traffic can surpass the network capacity in the monitored interface.

- *Distributed architecture*. On the other hand, a distributed architecture will perform multiple instances of Snort monitoring each one of the 'network units' composed by a PDC, the PMUs directly dependent on it, and the switch connecting the previous mentioned devices.
  In this case, Snort will monitor one interface of the switch that will have the rest traffic mirrored to it, making sure that we sniff all the packets in the network (we could just monitor the PDC, but we will lose a possible attempt of communication between PDCs, which should not happen under normal conditions). It is also worth noticing that we could simplify this architecture into monitoring only in the central switches of the network, instead of every single one of them, and then add more Snort instances as the network scales.
  However, this is not the objective of this research, and we will focus on the simple distributed architecture.
  The main advantage of this solution is the better scalability against the centralized one. Nevertheless, this architecture is more complex and prone to errors.

We have decided that the distributed architecture is the best for our needs, especially due to its scalability. After this technical decision, it is time to evaluate how the interaction between the Ryu controller and the Snort IDS will be performed. Again, we have two different options in this level of design:

- *Deploy Ryu and Snort on the same machine*. This option is more suitable for quick demos or tests. In this architecture, Ryu would receive Snort alerts packets via the Unix Domain Socket.

- *Deploy Ryu and Snort on different machines*. This option, on the other hand, has a better performance due to the large computational power that Snorts requires for analyzing packets. Ryu would receive Snort alerts packets via a Network Socket in this case.

We have decided that the most suitable solution is to run Ryu and Snort on different machines. We can see the proposed architecture in Fig. [2].

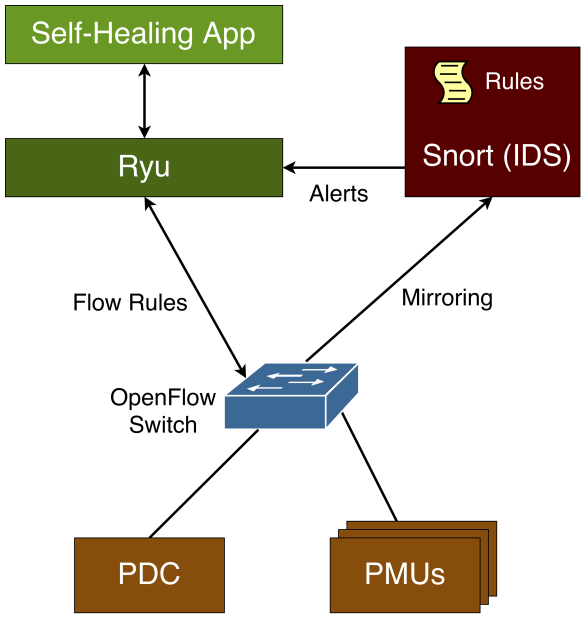

Fig. 2. Proposed architecture for the Ryu-Snort integration

The functionality is straightforward: in order to monitor packets between any hosts, Ryu will install the corresponding flows to mirror the packets to Snort, where the preprocessing and rule checking will take place. In case one of these rules is broken, Snort will send alerts to the SDN controller. After that, the controller application will decide how to proceed with the network self-healing process while keeping maximum observability.

C. Rules Development

One of the perks of integrating Snort with SDN is the flexibility that its rules offer. With a set of fine tuned rules, we can alert Ryu of several exploit attempts and act in consequence, either installing new SDN rules for the corresponding flows or starting the self-healing process if a PMU is compromised, for example. In this subsection we introduce a couple of rules as an example of how we can benefit of IDSs on SDN environments.

For example, we can write the following rules in order to alert the controller that a DoS attack is taking place in the network:

```
alert tcp any any -> $PDCIP any (msg:"Syn Flood to
     PDC"; flags:S,CE; flow:to_server;
    detection_filter: track by_src, count 500,
    seconds 1; priority:5; sid:1000001;)

alert tcp any any -> $PMUIP any (msg:'Syn Flood to
     PMU'; flags:S,CE; flow:to_server;
    detection_filter: track by_src, count 500,
    seconds 1; priority:5; sid:1000001;)
```

Specifically, the previous first rule will alert Ryu when any TCP connections from any IP address and any port with destination any port of the PDC surpass the rate of 500 packets per second, considered enough to interrupt the service of this network element. In the same way, the second rule will perform the an identical alert when the victim is any PMU.

This is just an example on how a simple rule can help detect and avoid a serious vulnerability. Literature on this topic is really scarce, but we can point out the work on developing rules against fuzzing attacks on [14].

There are multiple rules we can use for the Snort preprocessor in case we want to protect SCADA systems with Modbus or DNP3 protocols.

Suppose your network contains a DNP3 sensor device. In this scenario, the master station usually polls the device to read data, but it rarely sends any write requests to the device, as its normal operation is to send data to the master. If a write request is issued, this should be informed, as it may be an attack (or just legitimate behavior, but still anomalous). To inform of such an activity, you could use the following rule:

```
alert tcp $EXTERNAL_NET any -> $MY_SENSOR 20000 (
    msg:"WRITE REQUEST ON THE DNP3 SENSOR!"; flow:
    established,to_server; dnp3_func:write; sid
    :1000000;)
```

The previous rule would alert if a DNP3 Write request is seen going towards ”MY SENSOR”. If Snort is running inline, these requests can even be blocked.

You can see an example below:

```
drop tcp !$MASTER any -> $MY_SENSOR 20000 ( msg:"
    DNP3 Restart command received not originating
    from  Master device's IP address, It has been
    dropped fro security reasons"; flow:
    established,to_server; dnp3_func:cold_restart;
    sid:1000001;)
```

The previous rules is an example of how to make Snort automatically discard any packets. In this case, it will specifically drop packets containing a DNP3 command forcing a cold restart if it does not have the Master station's IP address as the source of the packet.

Snort is a very useful tool that makes it very easy to develop new rules, and is very flexible, making it a great choice as an Intrusion Detection System for our purpose.

D. Conclusions

We can extract a few conclusions about the integration of Snort in SDN networks:

- There is no perfect solution. In the design process we needed to overcome a few decisions regarding the architecture of our solution. Our election is not always optimal, but we believe is the most suitable one for our needs and requirements.

- The flexibility of Snort allows us to check for intrusion detections or generate alerts for other vulnerabilities in our system while keeping the benefits of SDN: data and control plane independence, ease of management, better network granularity.

- Additionally, SDN also benefits traditional IDS systems by adding that centralized view of the network provided by the controller, thus solving the management difficulties when we have multiple NIDS instances.

VII. Experiments

As mentioned before, the Snort IDS could either be integrated into the same device as the controller or run on a different machine. Below you can see both possibilities represented.

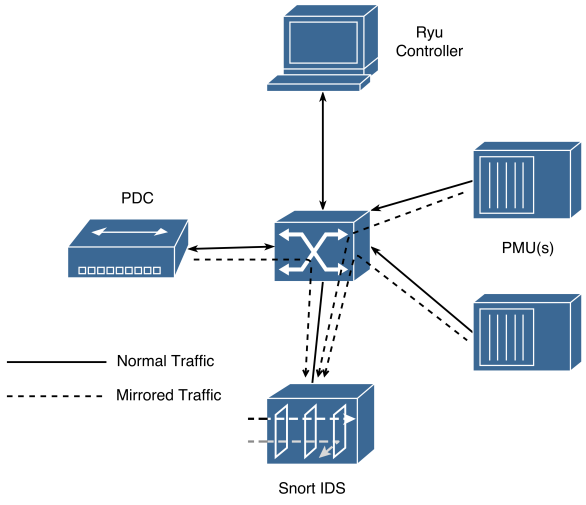

Fig. 3. SDN with Ryu and Snort IDS in separate machines

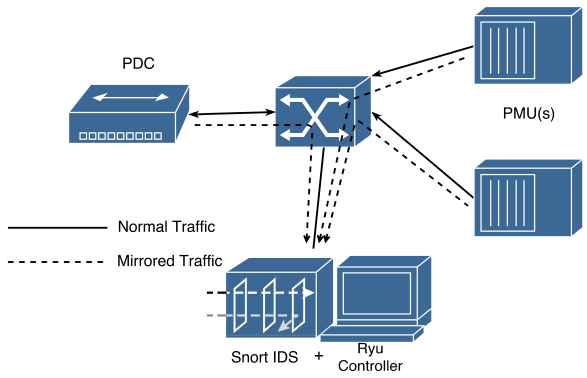

Fig. 4. SDN with Snort IDS integrated in the Ryu controller

We decided to use the second version for simplicity, as combining the Snort IDS and the controller gives an easier view of the network. It also avoids extra traffic exchanged between the IDS and the controller, because both are integrated into only one device.
In the following subsection we describe the topology used in the mininet environment to test the integration of Snort and Ryu.

A. *mininet scenario*

In the following picture you can have a glance at the network used in our tests, this is a screenshot from miniedit, the GUI tool developed for mininet.

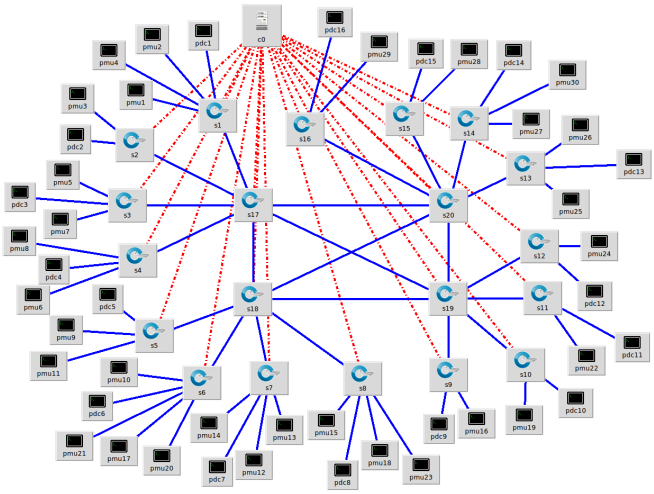

Fig. 5. mininet topology

The scenario shown in the above figure in miniedit is a simple PMU network consisting of:

- Controller: Ryu+ Snort IDS
- 4 backbone switches
- Multiple edge switches
- Each edge switch has a PDC connected
- Each edge switch has one or more PMUs connected
- Every PMU is assigned to a PDC, and traffic is only possible between PMUs and their assigned PDCs

B. *Running the complete network*

To simulate the network, several processes must be launched. The following steps have been taken to simulate the network:

- Run miniedit
- Load topology and run the network
- Wait for the network to be loaded into mininet. Start the developed controller which integrates Ryu and Snort IDS

Before testing the full network, the Ryu controller was tested independently in order to verify the correct operation of the controller and check the self-healing properties of the network. Once this is fully working, the Snort IDS can be run in the network, and exchange information with the Ryu controller.

- Start the Snort monitoring process
- At this point, everything is running in the same network, Snort IDS and Ryu are integrated into the controller

VIII. Results and Analysis

Using the sample network presented in the previous section, we tried to implement some Snort rules to validate the correct functioning of the implementation. In the following example we show some sample results taken from the tests. A rule was implemented to alert every time a ping was performed between a PDC and a PMU.

```
alertmsg: Pinging...
icmp(code=0,csum=9436,data=echo(data=array('B', [237, 129, 42, 87, 0, 0, 0, 0, 2
17, 4, 3, 0, 0, 0, 0, 0, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29,
 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49,
 50, 51, 52, 53, 54, 55]),id=8304,seq=3),type=8)
ipv4(csum=61060,dst='10.0.0.20',flags=2,header_length=5,identification=14352,off
set=0,option=None,proto=1,src='10.0.0.1',tos=0,total_length=84,ttl=64,version=4)
ethernet(dst='12:89:de:cd:2e:0d',ethertype=2048,src='3e:8d:01:a7:e5:bb')
```

Fig. 6. mininet topology

This alert is triggered by the following simple Snort rule:

```
alert icmp any any -> any any (msg:"Pinging...";
    sid:1000004;)
```

If we use a more specific rule, like blocking unwanted traffic trying to access the PMU from an external address, we can get a message like this:

```
alertmsg:  UNWANTED CONNECTION.
ipv4(csum=6674,dst='10.0.0.20'flags=2,header_length=5,identification=3242,offset
=0,option=None,proto=6,src='10.0.2.1',tos=0,total_length=40,ttl=64,version=4)
ethernet(dst='3e:8d:01:a7:e5:bb',ethertype=2048,src='12:89:de:cd:2e:0d')
```

Fig. 7. mininet topology

This would happen with an access attempt from an IP address outside the network 10.0.0.0/24 (in this case 10.0.2.1) if you use a rule like this:

```
alert ![10.0.0.0/24] any -> [10.0.0.0/24] 1111 (
    msg:" UNWANTED CONNECTION.";sid:1000005;)
```

After a real threat is detected, if the anomalous behavior comes from an internal device, it would be disconnected and isolated, and if it is a PDC, the PMUs previously connected to it would be redirected to a nearby PDC.

```
The failing PDC is 1

PMUs conected to this PDC are the following:
 PMU1: 10.0.0.20
 PMU2: 10.0.0.17
 PMU3: 10.0.0.18

Start self healing

New assigned PDC is PDC4


PMU with IP address 10.0.0.20 has been reconnected with PDC4

PMU with IP address 10.0.0.17 has been reconnected with PDC4

PMU with IP address 10.0.0.18 has been reconnected with PDC4

----------Routes fixed----------
```

Fig. 8. mininet topology

Here you can see the compromised PDC is ”PDC1”, so it was isolated. The controller checks which PMUs were sending its traffic to PDC1, and perform a self-healing algorithm to redirect their traffic to a nearby PDC, in this case PDC4.

The results of the tests are promising. The results combine success in different areas.

- First of all, Snort was deployed on the SDN Network and some rules have been successfully tested to produce alerts.
- The Self-healing mechanism has been tested and is working properly on the Ryu controller
- We were able to successfully integrate Snort and Ryu into the same network.In our test scenario we only tried to integrate Ryu and Snort into the same machine, but separating them into two different devices would be really similar, bu can prove to be more useful in a real environment were performance is key.

The custom Ryu controller integrating Snort successfully detected anomalies, leading to the disconnection of misbehaving PDCs, which can be marked as out of service.
Traffic from PMUs with destination to the compromised PDC will be redirected to a nearby PDC, allowing the network to self heal and maintain 100% observability, which is key in these critical networks.

IX. Conclusions

In this project we have presented an innovative approach to Intrusion Detection Systems in SCADA and WAMS Systems. This mechanism is supported by previous works in this area, which have been explained in Section II.

We believe that we have been able to successfully face the challenges that appeared in the design and development phases.
The developed system combines the strengths of two already existing technologies: the Ryu SDN controller and Snort, a widespread IDS open-source software. After monitoring the network and checking for the rules established, the IDS sends the corresponding alerts to the controller self-healing application.

The integration of Snort Intrusion Detection system with the Ryu SDN controller has proven to be successful in our experiments.

The use of this technology in critical infrastructure networks could possibly avoid some vulnerabilities these systems currently have, and protect the system against malicious traffic or cyber-attacks.

The combination of an Intrusion Detection system and an SDN network may prove very effective as it combines the flexibility and control advantages Software-Defined Networking bring on the table, while avoiding some of the vulnerabilities these networks have.

We believe this could lead to great improvements in efficiency and security for sensor networks.

X. Future Works

In this research project we have focused on the following topics:

- Implementation of self-healing in a ‘PMU-like’ SDN Network
- Implementation of the Intrusion Detection System Snort in an SDN Network
- Research on Snort Rules for PMU networks
- Integration of the Intrusion Detection System with the self-healing controller

Although we did integrate the self-healing controller and the Snort IDS into the same network, and we used a

sample PMU architecture, we did not use actual PMU host emulators, and therefore could not test the specific Snort rules for these environments. This would be a nice addition to the project.

REFERENCES

[1] P. Zhang, *"Phasor Measurement Unit (PMU) Implementation and Applications"*, Palo Alto, CA, 2007.

[2] T. Morris, S. Pan, J. Lewis, J. Moorhead, N. Younan, R. King, M.Freund, and V. Madani, *"Cybersecurity risk testing of substation phasor measurement units and phasor data concentrators"*, Proceedings of the Seventh Annual Workshop on Cyber Security and Information Intelligence Research - CSIIRW '11, 2011, pp. 14.

[3] C. Beasley, G. K. Venayagamoorthy, and R. Brooks, *"Cyber security evaluation of synchrophasors in a power system"*, Power SystemConference (PSC), 2014 Clemson University, 2014, pp. 15.

[4] Hui Lin, Chen Chen, Jianhui Wang, Junjian Qi, Dong Jin,*"Self-Healing Attack-Resilient PMU Network for Power System Operation"*

[5] A. A. Creery, E. J. Byres, *" Industrial Cybersecurity for power system and SCADA networks"*, IEE Industry Apllication Magazine, July-August 2007

[6] R. Chandia, J. Gonzalez, T. Kilpatrick, M. Papa and S. Shenoi, *"Security Strategies for Scada Networks"*,Proceeding of the First Int. Conference on Critical Infrastructure Protection, Hanover, NH., USA, March 19 - 21, 2007.

[7] M. Majdalawieh, F. Parisi-Presicce, D. Wijesekera, *"Distributed Network Protocol Security (DNPSec) security framework"*, In Proceedings of the 21st Annual Computer Security Applications Conference, December 5- 9,2005, Tucson, Arizona.

[8] I. Nai Fovino, M. Masera, R. Leszczyna, *"ICT Security Assessment of a Power Plant, a Case Study"*,Proceeding of the Second Int. Conference on Critical Infrastructure Protection, Arlington, USA, March 2008

[9] A. Carcano, I. Nai Fovino, M. Masera, A. Trombetta: *"Scada Malware, a proof of Concept"*,proceeding of the 3rd International Workshop on Critical Information Infrastructures Security, Rome, October 13-15, 2008

[10] Venkat Pothamsetty, Matthew Franz, *"Transparent Modbus/TCP Filtering with Linux"*,Available online http://modbusfw.sourceforge.net/ . Last access 03/10/2016

[11] M. Roesch, *"Snort -Lightweight Intrusion Detection for Networks"*,Proceedings of LISA 99: 13th Systems Administration Conference, Seattle, Washington, USA, November 712, 1999

[12] Andrea Carcano, Igor Nai Fovino, Marcelo Masera , *"Modbus/DNP3 State-based Filtering System"*, Institute for the Protection and Security of the Citizen Joint Research Centre European Commission Ispra, Italy

[13] IEEE, *"Standard for Synchrophasors for Power Systems"*, in IEEE Std C37.118-2005 (Revision of IEEE Std 1344-1995), 2006

[14] Read Sprabery, Thomas H. Morris, Shengyi Pan, Uttam Adhikari, and Vahid Madani, *"Protocol mutation intrusion detection for synchrophasor communications"*, In Proceedings of the Eighth Annual Cyber Security and Information Intelligence Research Workshop (CSIIRW '13), New York, NY, USA, 2013

[15] Stewart, John, et al., *"Synchrophasor security practices"*, Schweitzer Engineering Laboratories, Pullman, Washington, 2010

[16] Wireshark, https://www.wireshark.org/

[17] Yuan, Liwei, et al., *"Security breaches as PMU deviation: detecting and identifying security attacks using performance counters"*, Proceedings of the Second Asia-Pacific Workshop on Systems, ACM, 2011

[18] Sridhar, Siddharth, Adam Hahn, and Manimaran Govindarasu, *"Cyberphysical system security for the electric power grid"*, Proceedings of the IEEE 100.1, 2012

[19] Shepard, Daniel P., Todd E. Humphreys, and Aaron A. Fansler, *"Evaluation of the vulnerability of phasor measurement units to GPS spoofing attacks"*, International Journal of Critical Infrastructure Protection 5.3, 2012

[20] Chen, Jian, and Ali Abur, *"Placement of PMUs to enable bad data detection in state estimation"*, Power Systems, IEEE Transactions on 21.4, 2006

[21] Morris, T., et al., *"Cybersecurity testing of substation phasor measurement units and phasor data concentrators"*, The 7th Annual ACM Cyber Security and Information Intelligence Research Workshop (CSIIRW), 2011

[22] Sikdar, Biplab, and Joe H. Chow, *"Defending synchrophasor data networks against traffic analysis attacks"*, Smart Grid, IEEE Transactions on 2.4, 2011

[23] King, Joel W., *"Software-Defined Networking: Introduction to OpenFlow"* , 2014

XI. Appendix 1: mininet Network

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller, RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch, UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                         controller=RemoteController,
                         ip='127.0.0.1',
                         protocol='tcp',
                         port=6633)

    info( '*** Add switches\n')
    s17 = net.addSwitch('s17', cls=OVSKernelSwitch)
    s15 = net.addSwitch('s15', cls=OVSKernelSwitch)
    s20 = net.addSwitch('s20', cls=OVSKernelSwitch)
    s19 = net.addSwitch('s19', cls=OVSKernelSwitch)
    s16 = net.addSwitch('s16', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s11 = net.addSwitch('s11', cls=OVSKernelSwitch)
    s9 = net.addSwitch('s9', cls=OVSKernelSwitch)
    s8 = net.addSwitch('s8', cls=OVSKernelSwitch)
    s12 = net.addSwitch('s12', cls=OVSKernelSwitch)
    s13 = net.addSwitch('s13', cls=OVSKernelSwitch)
    s18 = net.addSwitch('s18', cls=OVSKernelSwitch)
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s14 = net.addSwitch('s14', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s10 = net.addSwitch('s10', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    pmu9 = net.addHost('pmu9', cls=Host, ip='10.0.0.25', defaultRoute=None)
    pmu12 = net.addHost('pmu12', cls=Host, ip='10.0.0.28', defaultRoute=None)
    pdc3 = net.addHost('pdc3', cls=Host, ip='10.0.0.3', defaultRoute=None)
    pdc11 = net.addHost('pdc11', cls=Host, ip='10.0.0.11', defaultRoute=None)
    pdc12 = net.addHost('pdc12', cls=Host, ip='10.0.0.12', defaultRoute=None)
    pmu27 = net.addHost('pmu27', cls=Host, ip='10.0.0.43', defaultRoute=None)
    pdc8 = net.addHost('pdc8', cls=Host, ip='10.0.0.8', defaultRoute=None)
    pmu14 = net.addHost('pmu14', cls=Host, ip='10.0.0.30', defaultRoute=None)
```

```
57      pmu19 = net.addHost('pmu19', cls=Host, ip='10.0.0.35', defaultRoute=None)
58      pdc15 = net.addHost('pdc15', cls=Host, ip='10.0.0.15', defaultRoute=None)
59      pdc4 = net.addHost('pdc4', cls=Host, ip='10.0.0.4', defaultRoute=None)
60      pmu22 = net.addHost('pmu22', cls=Host, ip='10.0.0.38', defaultRoute=None)
61      pdc5 = net.addHost('pdc5', cls=Host, ip='10.0.0.5', defaultRoute=None)
62      pmu5 = net.addHost('pmu5', cls=Host, ip='10.0.0.21', defaultRoute=None)
63      pmu10 = net.addHost('pmu10', cls=Host, ip='10.0.0.26', defaultRoute=None)
64      pmu4 = net.addHost('pmu4', cls=Host, ip='10.0.0.20', defaultRoute=None)
65      pmu25 = net.addHost('pmu25', cls=Host, ip='10.0.0.41', defaultRoute=None)
66      pmu13 = net.addHost('pmu13', cls=Host, ip='10.0.0.29', defaultRoute=None)
67      pdc9 = net.addHost('pdc9', cls=Host, ip='10.0.0.9', defaultRoute=None)
68      pmu17 = net.addHost('pmu17', cls=Host, ip='10.0.0.33', defaultRoute=None)
69      pmu28 = net.addHost('pmu28', cls=Host, ip='10.0.0.44', defaultRoute=None)
70      pmu6 = net.addHost('pmu6', cls=Host, ip='10.0.0.22', defaultRoute=None)
71      pdc6 = net.addHost('pdc6', cls=Host, ip='10.0.0.6', defaultRoute=None)
72      pmu1 = net.addHost('pmu1', cls=Host, ip='10.0.0.17', defaultRoute=None)
73      pmu8 = net.addHost('pmu8', cls=Host, ip='10.0.0.24', defaultRoute=None)
74      pmu29 = net.addHost('pmu29', cls=Host, ip='10.0.0.45', defaultRoute=None)
75      pmu15 = net.addHost('pmu15', cls=Host, ip='10.0.0.31', defaultRoute=None)
76      pmu16 = net.addHost('pmu16', cls=Host, ip='10.0.0.32', defaultRoute=None)
77      pdc7 = net.addHost('pdc7', cls=Host, ip='10.0.0.7', defaultRoute=None)
78      pmu26 = net.addHost('pmu26', cls=Host, ip='10.0.0.42', defaultRoute=None)
79      pmu30 = net.addHost('pmu30', cls=Host, ip='10.0.0.46', defaultRoute=None)
80      pdc2 = net.addHost('pdc2', cls=Host, ip='10.0.0.2', defaultRoute=None)
81      pdc1 = net.addHost('pdc1', cls=Host, ip='10.0.0.1', defaultRoute=None)
82      pmu23 = net.addHost('pmu23', cls=Host, ip='10.0.0.39', defaultRoute=None)
83      pdc14 = net.addHost('pdc14', cls=Host, ip='10.0.0.14', defaultRoute=None)
84      pmu21 = net.addHost('pmu21', cls=Host, ip='10.0.0.37', defaultRoute=None)
85      pmu11 = net.addHost('pmu11', cls=Host, ip='10.0.0.27', defaultRoute=None)
86      pdc16 = net.addHost('pdc16', cls=Host, ip='10.0.0.16', defaultRoute=None)
87      pdc10 = net.addHost('pdc10', cls=Host, ip='10.0.0.10', defaultRoute=None)
88      pmu7 = net.addHost('pmu7', cls=Host, ip='10.0.0.23', defaultRoute=None)
89      pdc13 = net.addHost('pdc13', cls=Host, ip='10.0.0.13', defaultRoute=None)
90      pmu20 = net.addHost('pmu20', cls=Host, ip='10.0.0.36', defaultRoute=None)
91      pmu3 = net.addHost('pmu3', cls=Host, ip='10.0.0.19', defaultRoute=None)
92      pmu24 = net.addHost('pmu24', cls=Host, ip='10.0.0.40', defaultRoute=None)
93      pmu2 = net.addHost('pmu2', cls=Host, ip='10.0.0.18', defaultRoute=None)
94      pmu18 = net.addHost('pmu18', cls=Host, ip='10.0.0.34', defaultRoute=None)
95
96      info( '*** Add links\n')
97      net.addLink(s1, s17)
98      net.addLink(s2, s17)
99      net.addLink(s3, s17)
100     net.addLink(s4, s17)
101     net.addLink(s5, s18)
102     net.addLink(s6, s18)
103     net.addLink(s7, s18)
104     net.addLink(s8, s18)
105     net.addLink(s19, s9)
106     net.addLink(s19, s10)
107     net.addLink(s11, s19)
108     net.addLink(s12, s19)
109     net.addLink(s16, s20)
110     net.addLink(s15, s20)
111     net.addLink(s14, s20)
112     net.addLink(s13, s20)
113     net.addLink(s17, s18)
114     net.addLink(s17, s20)
```

```
115         net.addLink(s20, s19)
116         net.addLink(s18, s19)
117         net.addLink(s17, s19)
118         net.addLink(s20, s18)
119         net.addLink(pdc2, s2)
120         net.addLink(pdc3, s3)
121         net.addLink(pdc4, s4)
122         net.addLink(pdc5, s5)
123         net.addLink(pdc6, s6)
124         net.addLink(pdc7, s7)
125         net.addLink(pdc8, s8)
126         net.addLink(pdc9, s9)
127         net.addLink(pdc10, s10)
128         net.addLink(pdc11, s11)
129         net.addLink(pdc12, s12)
130         net.addLink(pdc13, s13)
131         net.addLink(pdc14, s14)
132         net.addLink(pdc15, s15)
133         net.addLink(pdc16, s16)
134         net.addLink(pdc1, s1)
135         net.addLink(pmu2, s1)
136         net.addLink(pmu4, s1)
137         net.addLink(pmu1, s1)
138         net.addLink(s2, pmu3)
139         net.addLink(s3, pmu5)
140         net.addLink(s3, pmu7)
141         net.addLink(s4, pmu6)
142         net.addLink(s4, pmu8)
143         net.addLink(s5, pmu9)
144         net.addLink(s5, pmu11)
145         net.addLink(s6, pmu10)
146         net.addLink(s6, pmu17)
147         net.addLink(s6, pmu20)
148         net.addLink(s6, pmu21)
149         net.addLink(s7, pmu12)
150         net.addLink(s7, pmu13)
151         net.addLink(s7, pmu14)
152         net.addLink(s8, pmu15)
153         net.addLink(s8, pmu18)
154         net.addLink(s8, pmu23)
155         net.addLink(pmu16, s9)
156         net.addLink(s10, pmu19)
157         net.addLink(s11, pmu22)
158         net.addLink(s12, pmu24)
159         net.addLink(s13, pmu25)
160         net.addLink(s13, pmu26)
161         net.addLink(s14, pmu27)
162         net.addLink(s15, pmu28)
163         net.addLink(s16, pmu29)
164         net.addLink(pmu30, s14)
165
166         info( '*** Starting network\n')
167         net.build()
168         info( '*** Starting controllers\n')
169         for controller in net.controllers:
170             controller.start()
171
172         info( '*** Starting switches\n')
```

```
173         net.get('s17').start([c0])
174         net.get('s15').start([c0])
175         net.get('s20').start([c0])
176         net.get('s19').start([c0])
177         net.get('s16').start([c0])
178         net.get('s3').start([c0])
179         net.get('s4').start([c0])
180         net.get('s7').start([c0])
181         net.get('s6').start([c0])
182         net.get('s11').start([c0])
183         net.get('s9').start([c0])
184         net.get('s8').start([c0])
185         net.get('s12').start([c0])
186         net.get('s13').start([c0])
187         net.get('s18').start([c0])
188         net.get('s1').start([c0])
189         net.get('s14').start([c0])
190         net.get('s5').start([c0])
191         net.get('s10').start([c0])
192         net.get('s2').start([c0])
193
194         info('*** Post configure switches and hosts\n')
195
196         CLI(net)
197         net.stop()
198
199 if __name__ == '__main__':
200         setLogLevel('info')
201         myNetwork()
```

XII. Appendix 2: Controller

```
# Copyright (C) 2016 Jiaqi Yan, IIT
#
# Licensed under the Apache License, Version 2.0 (the "License");
# you may not use this file except in compliance with the License.
# You may obtain a copy of the License at
#
#     http://www.apache.org/licenses/LICENSE-2.0
#
# Unless required by applicable law or agreed to in writing, software
# distributed under the License is distributed on an "AS IS" BASIS,
# WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
# implied.
# See the License for the specific language governing permissions and
# limitations under the License.

"""
IEEE 30 Bus Power Network's Self-heal Controller
Based on an OpenFlow 1.0 L2 learning switch implementation.
"""

import array

from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_0
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.lib.packet import ipv4, icmp
from ryu.topology import event
from ryu.topology.api import get_switch, get_link
from ryu.lib import snortlib


failingPDCs=[False for i in range(0,16)]

class SelfHealController(app_manager.RyuApp):
    "SelfHeal controller based on ryu's simple switch"
    OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
    _CONTEXTS = {'snortlib': snortlib.SnortLib}


    def __init__(self, *args, **kwargs):
        """
        Create controller object

        Attributes:
            topology_api_app: monitor the topology changes by
                "ryu-manager --observe-links"
            switches(list of int): store every sw's dpid
            links(2-level dict): store the port number from
                sw(level-1 key, dpid) to sw(level-2 key, dpid)
            sw_to_host(dict): store the port number from sw(level-1 key, dpid)
```

```
57                 to host(level-2 key, IP string)
58         """
59         super(SelfHealController, self).__init__(*args, **kwargs)
60 
61         self.topology_api_app = self
62         self.switches = []
63         self.links = {}
64         self.sw_to_host = {}
65 
66         self.snort = kwargs['snortlib']
67         self.snort_port = 3
68         self.mac_to_port = {}
69         socket_config = {'unixsock': True}
70         self.snort.set_config(socket_config)
71         self.snort.start_socket_server()
72 
73         # init path from edge sw to PDCs
74         for i in range(1, 17):
75             self.sw_to_host[i] = {'10.0.0.%d' % i : 1}
76         # initial path from edge sw to PMUs
77         self.sw_to_host[1]['10.0.0.17'] = 4
78         self.sw_to_host[1]['10.0.0.18'] = 2
79         self.sw_to_host[1]['10.0.0.20'] = 3
80         self.sw_to_host[2]['10.0.0.19'] = 2
81         self.sw_to_host[3]['10.0.0.21'] = 2
82         self.sw_to_host[3]['10.0.0.23'] = 3
83         self.sw_to_host[4]['10.0.0.22'] = 2
84         self.sw_to_host[4]['10.0.0.24'] = 3
85         self.sw_to_host[5]['10.0.0.25'] = 2
86         self.sw_to_host[5]['10.0.0.27'] = 3
87         self.sw_to_host[6]['10.0.0.26'] = 2
88         self.sw_to_host[6]['10.0.0.33'] = 3
89         self.sw_to_host[6]['10.0.0.36'] = 4
90         self.sw_to_host[6]['10.0.0.37'] = 5
91         self.sw_to_host[7]['10.0.0.28'] = 2
92         self.sw_to_host[7]['10.0.0.29'] = 3
93         self.sw_to_host[7]['10.0.0.30'] = 4
94         self.sw_to_host[8]['10.0.0.31'] = 2
95         self.sw_to_host[8]['10.0.0.34'] = 3
96         self.sw_to_host[8]['10.0.0.39'] = 4
97         self.sw_to_host[9]['10.0.0.32'] = 2
98         self.sw_to_host[10]['10.0.0.35'] = 2
99         self.sw_to_host[11]['10.0.0.38'] = 2
100        self.sw_to_host[12]['10.0.0.40'] = 2
101        self.sw_to_host[13]['10.0.0.41'] = 2
102        self.sw_to_host[13]['10.0.0.42'] = 3
103        self.sw_to_host[14]['10.0.0.43'] = 2
104        self.sw_to_host[14]['10.0.0.46'] = 3
105        self.sw_to_host[15]['10.0.0.44'] = 2
106        self.sw_to_host[16]['10.0.0.45'] = 2
107        self.sw_to_host[17] = {}
108        self.sw_to_host[18] = {}
109        self.sw_to_host[19] = {}
110        self.sw_to_host[20] = {}
111 
112 
113    def packet_print(self, pkt):
114        pkt = packet.Packet(array.array('B', pkt))
```

```
115
116             eth = pkt.get_protocol(ethernet.ethernet)
117             _ipv4 = pkt.get_protocol(ipv4.ipv4)
118             _icmp = pkt.get_protocol(icmp.icmp)
119
120             if _icmp:
121                 self.logger.info("%r", _icmp)
122
123             if _ipv4:
124                 self.logger.info("%r", _ipv4)
125
126             if eth:
127                 self.logger.info("%r", eth)
128
129             # for p in pkt.protocols:
130             #     if hasattr(p, 'protocol_name') is False:
131             #         break
132             #     print('p: %s' % p.protocol_name)
133
134         def take_down_pdc(self):
135             print " keqerfqorgo "
136         @set_ev_cls(snortlib.EventAlert, MAIN_DISPATCHER)
137         def _dump_alert(self, ev):
138             msg = ev.msg
139
140             print('\n\nALERT: %s\n\nDetails:' % ''.join(msg.alertmsg))
141
142             self.packet_print(msg.pkt)
143             print "\n\n"
144
145         @set_ev_cls(event.EventSwitchEnter)
146         def _get_topology_data(self, ev):
147             """Automatically create sw to sw port table @self.links.
148
149             Notice that @ev is not used at all
150             """
151             self.logger.info("Updating port map between switches")
152             switch_list = get_switch(self.topology_api_app, None)
153             # create dpid to datapath mapping
154             for sw in switch_list:
155                 if sw.dp.id not in self.switches:
156                     self.switches.append(sw.dp.id)
157
158             # create links: the port on src sw to dst sw
159             link_list = get_link(self.topology_api_app, None)
160             for link in link_list:
161                 if link.src.dpid not in self.links.keys():
162                     self.links[link.src.dpid] = {}
163                 self.links[link.src.dpid][link.dst.dpid] = link.src.port_no
164             print self.links
165
166         @set_ev_cls(event.EventPortModify)
167         def _link_delete_handler(self, ev):
168             """React to link down event
169
170             @ev: from which we can extract Port object
171             """
172             port = ev.port
```

```
173         dpid = port.dpid
174         hosts=["no","no","no","no","no"]
175         hosts2=["no","no","no","no","no"]
176         discPMUs=["no","no","no","no","no"]
177         discPMUs2=["no","no","no","no","no"]
178         hosts3=["no","no","no","no","no"]
179         discPMUs3=["no","no","no","no","no"]
180         n=-1
181         dpid2=0
182         dpid3=0
183         global failingPDCs
184         failingPDCs[dpid-1]=True
185         print "\n\n----"+str(failingPDCs)+"----\n\n"
186         # port_no = port.port_no
187         # event_name = port.name
188
189         # make sure controller has global view
190         self._get_topology_data(None)
191
192         if port.is_down():
193             raw_input("Start self-healing by pressing Enter...")
194             print "The failing PDC is " + str(dpid)+"\n"
195             # path for pmu15(10.0.0.31) to pdc5(10.0.0.5)
196             hosts=self.sw_to_host[dpid].keys()
197             host=[]
198             host2=[]
199             size=len(hosts)
200             #print "\n Number of PMUs is: " + str(size)
201             print "PMUs conected to this PDC are the following:"
202             for num in range(0,size):
203                 host=map(int, hosts[num].split('.'))
204                 if hosts[num] !="no" and host[3]>16:
205                     n=n+1
206                     discPMUs[n]=hosts[num]
207                     print " PMU"+str(n+1)+": "+str(discPMUs[n])
208
209             print "\nStart self healing\n"
210             x=((dpid-1)/4)*4
211             y=x+3
212             coreswitch=17+x/4
213             #print "\n X= "+str(x)+"  Y= "+str(y)+"\n"
214             #print "\n ---Core switch is: " +str(coreswitch)+" ---\n"
215             newPDC=dpid-1
216             if newPDC==x:
217                 newPDC=y+1
218             print "New assigned PDC is PDC"+str(newPDC)+"\n"
219             #for now lets try to reconnect the pmus with the next pdc (pdc[dpid+1])
220             m=0
221             for m in range(0,n+1):
222                 #raw_input("Reconnect PMUs: PRess Enter..")
223                 self.sw_to_host[newPDC][discPMUs[m]] = self.links[newPDC][coreswitch]
224                 self.sw_to_host[coreswitch][discPMUs[m]] = self.links[coreswitch][dpid]
225                 self.sw_to_host[dpid]['10.0.0.'+str(newPDC)]=self.links[dpid][
                        coreswitch]
226                 self.sw_to_host[coreswitch]['10.0.0.'+str(newPDC)]=self.links[
                        coreswitch][newPDC]
227                 print "\nPMU with IP address "+discPMUs[m]+" has been reconnected with
                        PDC"+str(newPDC)#", IP: 10.0.0." + str(newPDC)
```

```
228
229                 #print "self.sw_to_host["+str(newPDC)+"]["+discPMUs[m]+"] = self.links
                        ["+str(newPDC)+"]["+str(coreswitch)+"]\n"
230                 #print "self.sw_to_host["+str(coreswitch)+"]["+discPMUs[m]+"] = self.
                        links["+str(coreswitch)+"]["+str(dpid)+"]\n"
231                 #print "self.sw_to_host[dpid]['10.0.0.'+str(newPDC)]=self.links[dpid][
                        coreswitch]"
232                 #print "self.sw_to_host[coreswitch]['10.0.0.'+str(newPDC)]=self.links[
                        coreswitch][newPDC]"
233
234                 if dpid==y+1:
235                     dpid2=x+1
236                 else:
237                     dpid2=dpid+1
238
239                 if dpid2==y+1:
240                     dpid3=x+1
241                 else:
242                     dpid3=dpid2+1
243
244                 #print " ---dpid: "+str(dpid)+"  dpid2: "+str(dpid2)+"---\n"
245
246                 if failingPDCs[dpid2-1] and  not failingPDCs[dpid3-1]:
247                     hosts2=self.sw_to_host[dpid2].keys()
248                     size2=len(hosts2)
249                     k=-1
250                     for num2 in range(0,size2):
251                         host2=map(int, hosts2[num2].split('.'))
252                         if hosts2[num2] !="no" and host2[3]>16:
253                                         k=k+1
254                                         discPMUs2[k]=hosts2[num2]
255                                         print " PMU"+str(k+1)+": "+str(discPMUs2[k
                                                ])
256
257                     m=0
258                     for m in range(0,k+1):
259                         self.sw_to_host[newPDC][discPMUs2[m]] = self.links[newPDC][
                                coreswitch]
260                         self.sw_to_host[coreswitch][discPMUs2[m]] = self.links[
                                coreswitch][dpid2]
261                         self.sw_to_host[dpid2]['10.0.0.'+str(newPDC)]=self.links[dpid2
                                ][coreswitch]
262                         self.sw_to_host[coreswitch]['10.0.0.'+str(newPDC)]=self.links[
                                coreswitch][newPDC]
263                         print "\nPMU with IP address "+discPMUs[m]+" has been
                                reconnected with PDC"+str(newPDC)#", IP: 10.0.0." + str(
                                newPDC)
264
265
266                 # elif failingPDCs[dpid2-1] and failingPDCs[dpid3-1]:
267
268
269             print "\n----------Routes fixed----------\n!"
270
271
272
273
274     def add_flow(self, datapath, in_port, dst, actions):
```

```
275         """Issue FlowMod message to switch @datapath
276
277         tell it that pkt to @dst should be send to @in_port.
278         @actions: list of PacketOutput actions(usually just one element)
279         """
280         ofproto = datapath.ofproto
281
282         match = datapath.ofproto_parser.OFPMatch(in_port=in_port, dl_dst=haddr_to_bin(
                dst))
283         #match = datapath.ofproto_parser.OFPMatch(in_port=in_port, nw_dst=dst)
284
285         mod = datapath.ofproto_parser.OFPFlowMod(
286             datapath=datapath, match=match, cookie=0,
287             command=ofproto.OFPFC_ADD, idle_timeout=0, hard_timeout=0,
288             priority=ofproto.OFP_DEFAULT_PRIORITY,
289             flags=ofproto.OFPFF_SEND_FLOW_REM, actions=actions)
290         datapath.send_msg(mod)
291
292     @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
293     def _packet_in_handler(self, ev):
294         """Ping(ICMP) packet handler"""
295         msg = ev.msg
296         datapath = msg.datapath
297         ofproto = datapath.ofproto
298         dpid = datapath.id
299
300         pkt = packet.Packet(msg.data)
301         pkt_eth = pkt.get_protocol(ethernet.ethernet)
302         eth_dst = pkt_eth.dst
303         eth_src = pkt_eth.src
304         # ignore lldp packet
305         if pkt_eth.ethertype == ether_types.ETH_TYPE_LLDP:
306             return
307         self.logger.info("packet in %s(port_%s) from %s to %s", \
308                          dpid, msg.in_port, eth_src, eth_dst)
309
310         pkt_icmp = pkt.get_protocol(icmp.icmp)
311         if pkt_icmp:
312             pkt_ip = pkt.get_protocol(ipv4.ipv4)
313             ip_dst = str(pkt_ip.dst)
314             # find path to dst host
315             if ip_dst in self.sw_to_host[dpid].keys():
316                 out_port = self.sw_to_host[dpid][ip_dst]
317                 self.logger.info("forward to port %s", out_port)
318                 # install a flow to avoid packet_in next time
319                 actions = [datapath.ofproto_parser.OFPActionOutput(out_port)]
320                 self.add_flow(datapath, msg.in_port, eth_dst, actions)
321                 #self.add_flow(datapath, msg.in_port, ip_dst, actions)
322                 data = None
323                 if msg.buffer_id == ofproto.OFP_NO_BUFFER:
324                     data = msg.data
325                 out = datapath.ofproto_parser.OFPPacketOut( \
326                                 datapath=datapath, buffer_id=msg.buffer_id, \
327                                 in_port=msg.in_port, actions=actions, data=data)
328                 datapath.send_msg(out)
329
330 #     @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
331 #     def _port_status_handler(self, ev):
```

```
332 #         """More general than EventPortModify?"""
333 #         msg = ev.msg
334 #         reason = msg.reason
335 #         port_no = msg.desc.port_no
336 #
337 #         ofproto = msg.datapath.ofproto
338 #         if reason == ofproto.OFPPR_ADD:
339 #             self.logger.info("port added %s", port_no)
340 #         elif reason == ofproto.OFPPR_DELETE:
341 #             self.logger.info("port deleted %s", port_no)
342 #         elif reason == ofproto.OFPPR_MODIFY:
343 #             self.logger.info("port modified %s", port_no)
344 #         else:
345 #             self.logger.info("Illeagal port state %s %s", port_no, reason)
```