Network Traffic Monitor for IDS in IoT

Diego Angelo Bolatti [1] [0000-0002-8275-4476], Carolina Todt[1] [0000-0001-8429-6141], Reinaldo Scappini[1] [0000-0001-6854-4643] and Sergio Gramajo[1] [0000-0001-5091-7931]

[1] Center for Applied Research in Information and Communication Technologies at National University of Technology (UTN), Resistencia Regional Faculty (UTN-FRRe). French St. 414, Resistencia, Province of Chaco, Argentina.
{dbolatti, carolinatodt, rscappini, sergiogramajo}@gfe.frre.utn.edu.ar

Abstract. As network services and IoT technologies rapidly evolve, in literature there are many anomalies detection proposals based on datasets to deal with cybersecurity threats. Most of this proposal uses structured data classification and they can recognize with a certain degree of accuracy whether a type of traffic is "anomalous" or not. Even what kind of anomaly it has. Nevertheless, previous works do not clearly indicate the technical methodology to set up the data gathered scenarios. As a main contribution, we are going to show a detailed deployment IoT traffic monitor ready for intelligent network traffic classification. Monitoring and sniffers are an essential concept in network management as it helps network operators to determine the network behavior and status of its components. Anomaly detection also depends on monitoring for decision-making. Thus, this paper will describe the creation of a portable network traffic monitor for IoT using Docker container and bridge networking with SDN.

Keywords: Network Monitoring, IoT, IDS, SDN.

1 Introduction

Nowadays, as technology becomes more widely available, millions of users worldwide have used some type of smart device. The number of smart homes in Europe and North America has reached 102.6 million in 2020 and it will be 179 million in 2024 [1]. At the same time more sophisticated IoT applications are deploying, and they use devices, sensors, smart techniques to bring information or knowledge or, even, make decisions [2, 3]. In a broad sense, this concept is Internet of Things (IoT) [4, 5] that was the result of conventional network evolution connecting millions of devices with minimal human intervention to later make any kind of decisions [6-8].

This has left the IoT vulnerable to various types of security threats just like other technologies [9, 10]. In an effort to address these issues, different Intrusion Detection Systems (IDS) techniques have been proposed. Currently anomaly-based network intrusion detection is an important field of research [11, 12].

In this way, Intrusion Detection Systems analyze network traffic to detect malicious behavior. For its deployment it is necessary (i) Collect information; (ii) Analyze the

information; (iii) Identify threats or normal traffic through security events; and (iv) Detect and report threats [13]. The implementation described in this paper is focused only on point (i).

Many times, free open-source network sniffers are used to capture network traffic data and then, this data is labeled as a type of attack or normal traffic in an off-line way with datasets. Different types of intelligent approaches have been used like Machine Learning [14-16] and Deep Learning [17-22] in order to identify and classify threats. However, there is a lack of information about how an efficient IoT-based datasets scenario is obtained [23–29].

This work will not cover the complete development of an intelligent anomaly detection system for IoT, here we will show the theoretical fundaments and the basic elements to create a scenario to collect information from the IoT infrastructure, elaborated as the first part of the research project called "Intelligent Anomaly Detection System for IoT" [30] and part of the Technical Report "Proposal" presented at the International Telecommunications Union [31].

The remainder of the paper is structured as follows: Network Traffic Monitor Architecture is introduced in Section 2. A detailed description of deployment proposed is given in Section 3. Section 4 introduces the creation of an SDN Controller and traffic gathering. The conclusions and future work are given in Section 5

2 Network Traffic Monitor Architecture

In principle, it is necessary to define the scope in which the proof of concept is created. Fig 1 shows the proposed architecture in four layers: device layer with Software Defined Network (SND) switch [32] and gateway with Openflow monitor where our proposal is deployment.

To design the monitoring system, we base our work on the following 3 concepts:

Namespace: a namespace in computer science is an abstract container or environment created to hold a logical grouping of unique identifiers or symbols (i.e. names).

Docker: is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux [33]. Docker uses the resource isolation features of the Linux kernel such as *cgroups* and kernel namespaces, and a union-capable file system such as aufs and others to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines. The Linux kernel's support for namespaces mostly isolates an application's view of the operating environment, including process trees, network, user IDs and mounted file systems, while the kernel's *cgroups* provide resource limiting, including the CPU, memory, block I/O and network. Since version 0.9, Docker includes the *libcontainer* library as its own way to directly use virtualization facilities provided by the Linux kernel, in addition to using abstracted virtualization interfaces via *libvirt*, LXC (Linux Containers) and *systemd-nspawn*. Docker perfectly adjusts the needs of

our work, because it can be implemented in the same way, both in a testing and simulation environment within a virtual machine, or in a production environment, on local servers, cloud servers, etc.

Connectivity Network with Docker: when installing Docker on an operating system, it creates a bridge to a network called *docker0*, and this network connects by default all containers, unless otherwise indicated. If a virtual machine with Docker is used, it is necessary to configure the network understanding that it is necessary to work with different levels of abstraction. In the case of containers (they have their own namespace), it will be necessary to connect them to the virtual machine where they are running, and this, in turn, to the host, to the Internet, and/or to other virtual machines if they exist. Figure 2 shows in the shaded box, the concrete boundaries of the implementation of the access module to the proposed IoT architecture. It is then understood that it is implemented in the operating system that supports the IoT Gateway and the traffic switching and monitoring module.

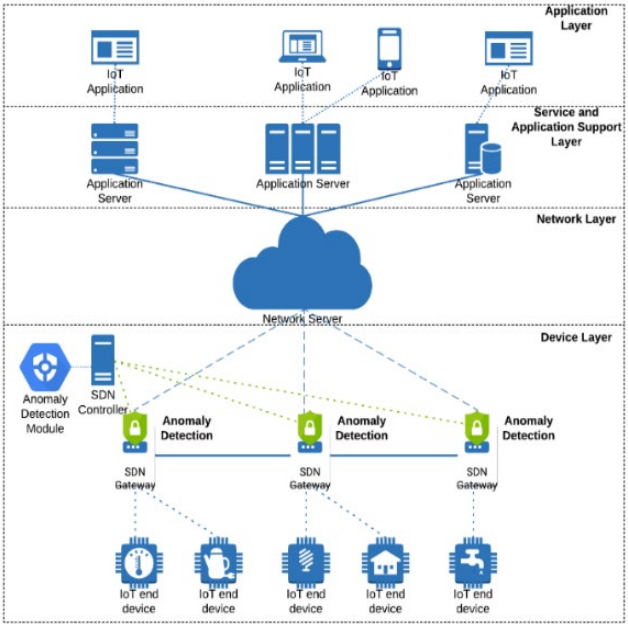


Fig. 1. Architecture of the intelligent anomaly detection system for IoT.

The implementation of the switching and traffic monitoring module is described in detail below, highlighting that the implementation makes no difference, whether it is carried out in a virtual machine environment or in a production environment directly on the host operating system (usually Linux). This is due to the characteristic of isolation that namespaces have, which allows a great portability. Another issue in this design is that the SDN controller and the Machine Learning module have absolute independence in terms of their physical location; it is enough to properly define the corresponding communication channel. Basically, the scenario in which we worked to realize the

design and testing of the proposed architecture is a computer that we call a base machine, with an O.S. Ubuntu 20.04 and a virtualization system VirtualBox Version 6.1.10_Ubuntu r138449. In addition, we use a Virtual Machine (VM) named "*iot*" configured with Ubuntu 20.04 OS.

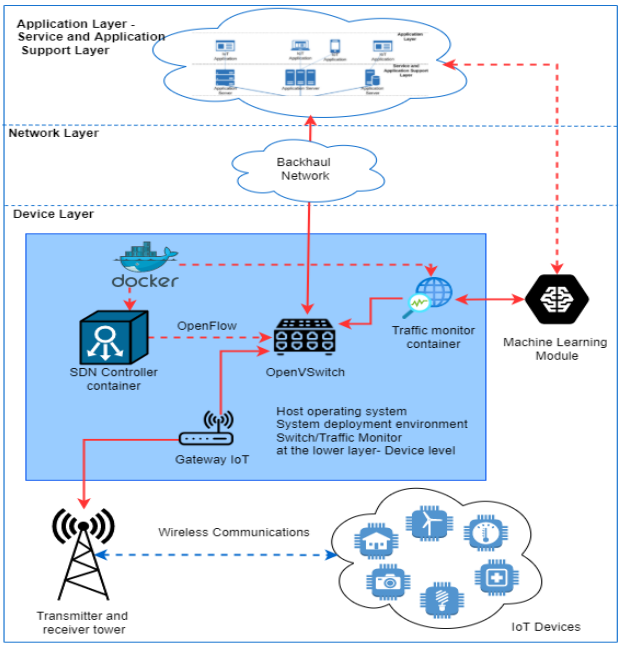


Fig. 2. Diagram of the Monitoring system.

3 Deployment and Testing

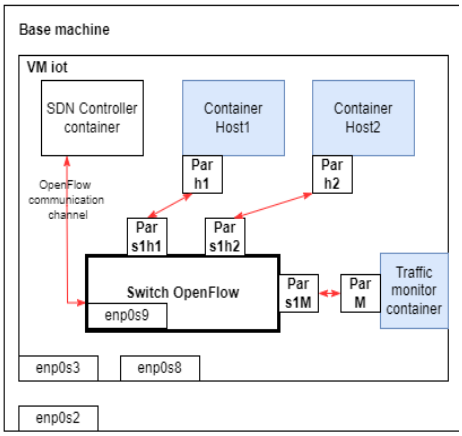
To test the components of the traffic reception and monitoring system, the following software is installed on the VM iot:

- Docker *version 19.03.13, build 4484c46d9d.*
- Open *vSwitch version 2.13.0*

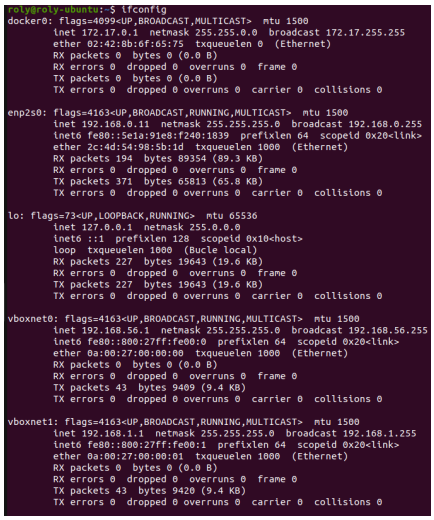
With this software, you can use containers and manage traffic with an OpenFlow switch (OpenvSwitch). The outline of the study scenario is shown in Figure 3 (a).

The components of the study topology comprise an SDN controller, an OpenFlow switch that performs the functions of managing traffic, two hosts called Host1 and Host2, which are intended for different connectivity tests, and a container called Monitor, which includes the appropriate software to capture traffic. The base machine has a physical Ethernet interface called *enp0s2* and a network of the VirtualBox hypervisor, *vboxnet0* and *vboxnet1* respectively (Figure 3. (b)).

Four networks are defined, three of which are managed by the VirtualBox hypervisor and one by Docker. A bridge type network with *enp0s3* interface linked to the *enp0s2* interface of the base machine, this allows sending and receiving traffic to and from all the VM devices. A connection to the *vboxnet0 192.168.56.0/24* network with a host-only interface (called *enp0s8* on the VM). A *vboxnet1 192.168.1.0/24* network connection with a host-only interface (called enp0s9 in the VM). Docker manages by default a network that allows it to connect the containers by assigning ip addresses using *dhcp* on the *172.17.0.0/24* network. Initially, we check the status of the interfaces in the VM iot.



a. Test Scenario.



b. Initial configuration of VM iot.

Fig. 3. Deployment scenario.

Linux supports individual network namespaces, each host has a default *netns*, named by default, and each host container is isolated into its *netns* that are identified by an integer.

In the next subsections we will review in detail the important preliminary aspects before creating the monitor.

3.1 Creating topology elements. OpenFlow Switch

The next step is to configure the *ovs* Linux switch by executing the following commands:

sudo ovs-vsctl add-br s1
sudo ifconfig enp0s9 0
sudo ovs-vsctl add-port s1 enp0s9
sudo ifconfig s1 192.168.1.11/24

With these commands an OpenFlow switch called *s1* was created. To which the interface *enp0s9* was added and *s1* was configured with the ip *192.168.1.11/24*.

3.2 Creating links between components

To make the corresponding connections between the different components of our scenario we will make use of a feature provided by the Linux kernel called virtual Ethernet link (*veth*), these virtual Ethernet devices can act as tunnels between network namespaces, or they are used to create a bridge to a physical environment with a network device present in another namespace, they can also be used as independent network devices. *Veth* devices are always created in interconnected pairs, it is an analogy to an ethernet cable with two ends. Packets transmitted on a device at one end of the pair are immediately received at the device at the other end. When either device is inactive, the link state of the peer is inactive. Three *veth* are added corresponding to *Host1, Host2*, and *Monitor* respectively (see Fig. 4):

1. One end named *par_s1h1* for the switch and one end named *par_h1* for *Host1*.
2. One endpoint named *par_s1h2* for the switch and one endpoint named *par_h2* for *Host2*.
3. An endpoint named *par_s1M* for the switch and an endpoint named *par_M* for the Monitor.

```
iot@iot:~$ sudo ip link add par_s1h1 type veth peer name par_h1
iot@iot:~$ sudo ip link add par_s1h2 type veth peer name par_h2
iot@iot:~$ sudo ip link add par_s1M type veth peer name par_M
iot@iot:~$ sudo ip link ls
```

Fig. 4. Ethernet links of the global space.

The endpoints of the corresponding virtual links created are incorporated into switch s1 as ports and the interfaces are up (Figure 5 (a)).

```
iot@iot:~$ sudo ovs-vsctl add-port s1 par_s1h1
iot@iot:~$ sudo ovs-vsctl add-port s1 par_s1h2
iot@iot:~$ sudo ovs-vsctl add-port s1 par_s1M
iot@iot:~$ sudo ip link set par_s1h1 up
iot@iot:~$ sudo ip link set par_s1h2 up
iot@iot:~$ sudo ip link set par_s1M up
iot@iot:~$ sudo ovs-vsctl show
806347cd-1c85-4833-a1bb-b5c357bc6964
    Bridge s1
        Port enp0s9
            Interface enp0s9
        Port s1
            Interface s1
                type: internal
        Port par_s1M
            Interface par_s1M
        Port par_s1h1
            Interface par_s1h1
        Port par_s1h2
            Interface par_s1h2
    ovs_version: "2.13.0"
```

a. Switch with ports added.

```
iot@iot:~$ sudo docker run -it --network=none --name h1 4b7a9ac93bca
[sudo] password for iot:
/ # ifconfig
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

/ #
```

b. Command line container *Cont1*.

```
iot@iot:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
14cb73456d20        4b7a9ac93bca        "/bin/sh"           41 seconds ago      Up 37 seconds                           monitor
```

c. Docker *ps* command.

Fig. 5. Commands.

Next, we will create the other elements we need within the topology, using Docker. Then, from the local Docker repository images in the VM, we start a container named 1.0 to monitor the traffic. To obtain the configuration of the monitor interfaces, run the following command (Figure 5 (b)).

3.3 Connecting the monitor

We open another terminal in the VM and obtain the PID of the container, with the *inspect* command of Docker, which returns low-level information of the Docker objects. In order to do this, the list of processes in Docker is queried with the docker *ps* command (Figure 5 (c)).

Then ***docker inspect -f '{{.State.Pid}}' 14cb73456d20***. returns as the result: *2051*. To move the par_M endpoint into the monitor namespace *run:**sudo ip link set par_M netns 2051*** . Then we execute the following command: ***sudo ln -s /proc/2051/ns/net /var/run/netns/2051***. Then directly from the VM terminal we can comfortably do ip netns exec (namespace) (command to execute in the namespace). In our case: ***sudo ip netns exec 2051 ip link list*** (Figure 6).

```
iot@iot:~$ sudo ln -s /proc/2051/ns/net /var/run/netns/2051
iot@iot:~$ sudo ip netns exec 2051 ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
12: par_M@if13: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether aa:3a:24:95:12:a9 brd ff:ff:ff:ff:ff:ff link-netnsid 0
iot@iot:~$
```

Fig. 6. Virtual link to the directory containing the namespace monitor.

We continue the scenario assembly by assigning in the monitor container the name eth1 to the end par_M. Then we assign as interface name eth1 the par_M name on the monitor as follows: ***sudo ip netns exec 2051 ip link set dev par_M name eth1***, the interface is activated: ***sudo ip netns exec 2051 ip link set eth1 up***. In the monitor namespace you can check the result of these last two actions as shown in Figure 7.

```
iot@iot:~$ sudo ip netns exec 2051 ifconfig
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        ether aa:3a:24:95:12:a9  txqueuelen 1000  (Ethernet)
        RX packets 9  bytes 726 (726.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Fig. 7. Verifying network configuration in the monitor container.

In the monitor assign to eth1 an ip address of the network connected to the ovs switch (which in our case is the vboxnet1 192.168.1.0/24 network): ***sudo ip netns exec 2051 ifconfig eth1 192.168.1.221/24*** then verify again in the monitor (Figure 7).

```
iot@iot:~$ sudo ip netns exec 2051 ifconfig eth1 192.168.1.221/24
iot@iot:~$ sudo ip netns exec 2051 ifconfig
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.1.221  netmask 255.255.255.0  broadcast 192.168.1.255
        ether aa:3a:24:95:12:a9  txqueuelen 1000  (Ethernet)
        RX packets 12  bytes 936 (936.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

lo: flags=73<UP,LOOPBACK,RUNNING>  mtu 65536
        inet 127.0.0.1  netmask 255.0.0.0
        loop  txqueuelen 1000  (Local Loopback)
        RX packets 0  bytes 0 (0.0 B)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 0  bytes 0 (0.0 B)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

Fig. 7. Assigning ip to the monitor interface.

As you can see the monitor now has an ethernet interface with the name *eth1*, the ip *192.168.1.221/24* and is connected to the switch port named *par_s1M*. At this point, we have added an OpenFlow switch named *s1* and a container named monitor into the scenario and connected them together.

3.4 Creating Host 1 and Host 2

First we create the containers Host1 and Host2, based on the Linux Alpine image we have available in the local repository with the *ID:a24bb4013296:* ***docker run -itd --name=cont1 --net=none a24bb4013296***.

3.5 Connecting Host 1 and Host 2

As can be seen in the figure 8, the containers were created and were running in the background.

```
iot@iot:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
e334b86f2bf9        a24bb4013296        "/bin/sh"           6 minutes ago       Up 5 minutes                            cont2
d905788711ef        a24bb4013296        "/bin/sh"           6 minutes ago       Up 6 minutes                            cont1
14cb73456d20        4b7a9ac93bca        "/bin/sh"           About an hour ago   Up About an hour                        monitor
iot@iot:~$
```

Fig. 8. Processes running in Docker.

At this point we can find out the respective namespaces (Figure 9).

```
iot@iot:~$ docker inspect -f '{{.State.Pid}}' d905788711ef
2371
iot@iot:~$ docker inspect -f '{{.State.Pid}}' e334b86f2bf9
2420
```

Fig. 9. Docker inspect command.

As shown in figure 9, the Host1 namespace is *2371*, and the *Host2* namespace is *2420*.

- To move the par_h1 endpoint into the Host1 namespace, you run: ***sudo ip link set par_h1 netns 2371***.
- To move the par_h2 endpoint into the Host2 namespace run: ***sudo ip link set par_h1 netns 2420***.

Next, virtual links are created to the /proc directory:

sudo ln -s /proc/2371/ns/net /var/run/netns/2371
sudo ln -s /proc/2420/ns/net /var/run/netns/2420

We proceed to verify that the endpoints are in the respective spaces (Figure 10).

```
iot@iot:~$ sudo ip netns exec 2371 ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
8: par_h1@if9: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 9a:7e:6c:33:18:3c brd ff:ff:ff:ff:ff:ff link-netnsid 0
iot@iot:~$ sudo ip netns exec 2420 ip link list
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default qlen 1000
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
10: par_h2@if11: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
    link/ether 86:55:ac:96:7f:b4 brd ff:ff:ff:ff:ff:ff link-netnsid 0
iot@iot:~$
```

Fig. 10. Verification of links on Host1 and Host2 netns.

As shown in the figure, par_h1 is in the *Host1* space, and *par_h2* in the *Host2* space. Then we proceed to give them name and interface (*eth1*) corresponding ip address and raise them to be active and set the corresponding ip. The complete sequence is shown in Figure 11.

```
iot@iot:~$ sudo ip netns exec 2371 ip link set dev par_h1 name eth1
iot@iot:~$ sudo ip netns exec 2420 ip link set dev par_h2 name eth1
iot@iot:~$ sudo ip netns exec 2371 ip link set eth1 up
iot@iot:~$ sudo ip netns exec 2420 ip link set eth1 up
iot@iot:~$ sudo ip netns exec 2371 ifconfig eth1 192.168.1.222/24
iot@iot:~$ sudo ip netns exec 2420 ifconfig eth1 192.168.1.223/24
iot@iot:~$ sudo ip netns exec 2371 ifconfig
```

Fig. 11. Host1 and Host2 Network Configuration.

4 Creating SDN Controller and Traffic Sniffer

Now it is time to add the SDN controller to the topology and connect it to the switch to manage it, for this we have in the local docker repository an image with a version of the onos SDN controller, with *ID: c07e43df3bf2*.

We execute the following command to run the SDN controller and access to its interface: ***docker run -itd -p 6653:6653 -p 8181:8181 -p 8101:8101 -p 5005:5005 -p 830:830 --name=onos c07e43df3bf2***.

Once logged in onos, two applications are added: Open Flow Provider Suite and Reactive Forwarding. Once the controller is running, the switch is connected to it in order to be able to install flow rules through it. Tell the switch s1 which version of OpenFlow it is going to operate with: ***sudo ovs-vsctl set bridge s1 protocols=OpenFlow13*** and connect switch s1 to the controller using: ***sudo ovs-vsctl set-controller s1 tcp:192.168.56.11:6653***.

Keep in mind that 192.168.56.0 is the vboxnet0 network and thanks to have configured a bridge network in VirtualBox, we can have communication with all networks that we have configured in the MV iot, thanks to this also works port forwarding docker and we can use the browser of the base machine to access them. Once this point is reached, you can operate the switch at low level via terminal with the commands provided by *ovs-vsctl* and *ovs-ofctl*.

Traffic Capture Function. OVS provides a way to duplicate network traffic from specific ports to a dedicated outgoing port. The duplication can be in one direction or both. The following are the commands to create a port that shows traffic and connect it to the

monitor container for processing purposes. First, create the mirror port on the switch (Figure 12).

```
iot@iot:~$ sudo ovs-vsctl -- --id=@m create mirror name=espejo -- add bridge s1 mirrors @m
[sudo] password for iot:
71940157-4e43-4bc3-846a-2440d851fa06
```

Fig. 12. Inserting mirror port on the switch.

Next, we will get the uuids of the ports we are interested in for the switch configuration by mirroring the desired traffic to the monitor port (Figure 13).

```
iot@iot:~$ sudo ovs-vsctl get port "enp0s9" _uuid
5cbd9cd2-8164-42dd-95b9-5bb07085c2a0
iot@iot:~$ sudo ovs-vsctl get port "par_s1h1" _uuid
9fdd1399-2d78-455a-a50a-bb61b3e994db
iot@iot:~$ sudo ovs-vsctl get port "par_s1h2" _uuid
5b8ee856-1b28-4e4d-8dcf-122bcced7d6f
iot@iot:~$ sudo ovs-vsctl get port "par_s1M" _uuid
b9e505e4-5694-4f0c-b4f7-39209a5122e1
iot@iot:~$
```

Fig. 13. Obtaining the uuid of the switch s1 ports.

With this information we can configure which are the ports whose inbound and/or outbound traffic we want to show. To do this we execute the following command:
sudo ovs-vsctl set mirror espejo select_src_port=9fdd1399-2d78-455a-a50a-bb61b3e994db,5b8ee856-1b28-4e4d-8dcf-122bcced7d6f select_dst_port=9fdd1399-2d78-455a-a50a-bb61b3e994db,5b8ee856-1b28-4e4d-8dcf-122bcced7d6f

With the above command we inform the switch that we want all traffic exchange in both directions from the ports that are connected to Host1 and Host2. We verify the configuration as follows: ***sudo ovs-vsctl list mirror mirror***.

Now all that remains is to inform which port will be the outgoing port for the duplicated traffic using the command: *ovs-vsctl -- --id=@(*uuid corresponding to the output) get port (uuid corresponding to the output) -- *set mirror mirror output-port=@(*uuid corresponding to the output)
sudo ovs-vsctl -- --id=@b9e505e4-5694-4f0c-b4f7-39209a5122e1 get port b9e505e4-5694-4f0c-b4f7-39209a5122e1 -- set mirror mirror output-port=@b9e505e4-5694-4f0c-b4f7-39209a5122e1

There you can see the outgoing port for the duplicated traffic, which is the port that corresponds to the connection with the monitor. Under these conditions, we can test generating traffic between Host1 and Host2, and see if we can capture it in the monitor container, for that we run in a terminal the command ***docker* ps** to identify the process where the monitor is running and then with the command ***docker exec -it (process)* sh**, returns us a terminal with command line inside the monitor container and once there we configure a data capture with tcpdump (Figure 14).

```
iot@iot:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
                                        NAMES
c9b7338c55f8        c07e43df3bf2        "./bin/onos-service …"   3 hours ago         Up 3 hours
/tcp, 0.0.0.0:8181->8181/tcp, 9876/tcp  onos
e334b86f2bf9        a24bb4013296        "/bin/sh"                4 hours ago         Up 4 hours
                                        cont2
d905788711ef        a24bb4013296        "/bin/sh"                4 hours ago         Up 4 hours
                                        cont1
14cb73456d20        4b7a9ac93bca        "/bin/sh"                5 hours ago         Up 5 hours
                                        monitor
iot@iot:~$ docker exec -it 14cb73456d20 sh
/ # tcpdump -c 100 -w trafh1h2.pcap
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size 262144 bytes
```

Fig. 14. Configuration of traffic capture inside the monitor container.

As you can see, a capture of 100 frames was configured and saved in a file named trafh1h2.pcap. We also do the same by opening a terminal on h1 and pinging h2 to generate a sample traffic (Figure 15).

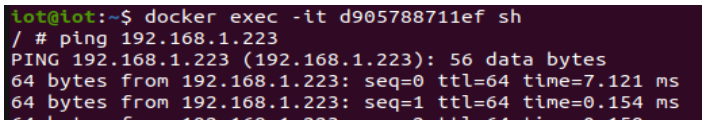


Fig. 15. Generating traffic from Host1 to Host2.

This generates continuous traffic between *cont1* and *cont2*, and in the monitor container a file is generated containing the capture of the traffic between *cont1* and *cont2*, reflected in the port of connection of the monitor to the OpenFlow switch as shown in Figure 16.

Fig. 16. Capture file generated on the monitor.

Once the capture file is generated we copy it from the container to the MV iot with the command: ***docker cp monitor:/trafh1h2.pcap***. Then from the MV iot to the base machine in the Documents folder to be able to analyze it with Wireshark with: ***scp iot@192.168.56.11:/home/iot/trafh1h2.pcap Documents/*** . And finally, we open the file with the capture to examine it (Figure 17).

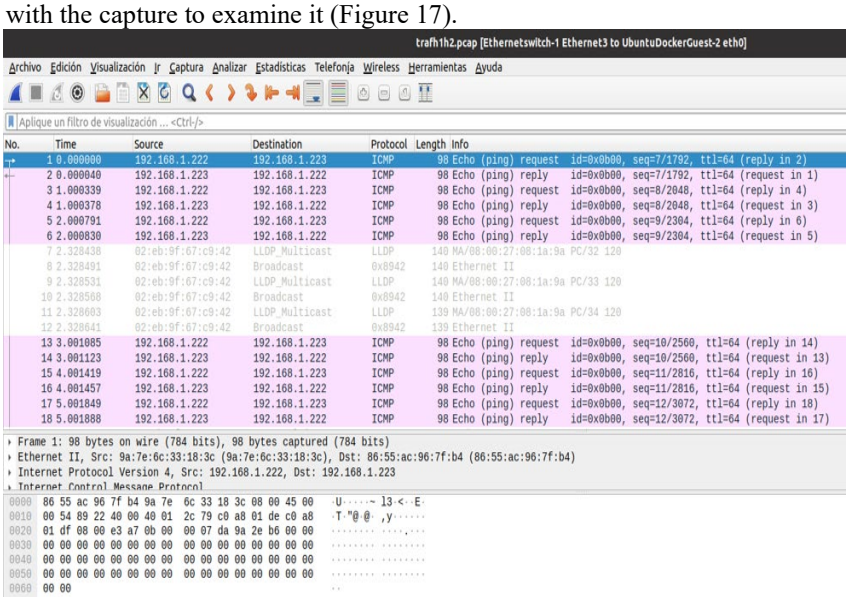


Fig. 17. Wireshark screen displaying the file with the data capture performed.

As shown in the figure, we have the traffic of the selected ports, which was captured by the monitor container. In the same way, we can proceed to select any interface and type of traffic because the design is flexible to adapt to any scenario and has the advantage of being “portable”. The example traffic between two generic hosts was chosen, to represent the input and output of backbone traffic, as it could perfectly be representing the output of the IoT Gateway in its transit to the network backhaul. This work describes all the low-level engineering to build the scenario, analyzing all the functional components and connectivity.

5 Conclusions and future work

In this work, we show the design and implementation of an IoT network monitoring system that provides network traffic data and statistics for the top layer of the IoT architecture. The results of the experiment show the feasibility of the traffic monitoring system and its application in the device layer of the IoT architecture. As a tutorial, it has been shown step by step how to create an architecture for data capture with an IoT platform based on traffic analyzers and SDN from a scenario divided into abstraction layers. This work is the baseline for the collection of robust data that will later become part of IDS and learning methods for network traffic classification.

It is essential to mention that this work deals with the study and implementation of the capture module, from the lowest level acting directly on the linux kernel, it aims to show in a didactic way, acquire the "know how" to understand higher level developments and with greater ease of implementation, offering a fully modular and scalable solution with the possibility of using orchestration tools, such as Kubernetes, Terraform, etc. We want to highlight the introductory nature of this document based on this objective.

In accordance with the above, with a view to continuing this work, and with the aim of improving the performance of the traffic monitor shown here, we will soon publish the progress we made in the design of a monitoring module which a concept of "promiscuous bridge", designed from a bridge and a Docker container that contains the capture software tcpdump with all the advantages of capturing in "raw" mode, capturing all the packets of a given interface reflected in the bridge interface at which the monitor is connected. By capturing all the traffic of the chosen interface, the bridge also allows multiple tools to obtain the same data, which is very useful; if, for example, you want to define traffic selection and filter functions, a fundamental requirement for the development of an IDS. It should be noted that the use of Docker provides an isolated and easily replicable environment that ensures portability and implementation of the monitor wherever it is needed.

References

1. Berg Insight: IoT Business News, https://iotbusinessnews.com/2021/02/11/06951-the-number-of-smart-homes-in-europe-and-north-america-will-reach-179-million-in-2024/.

2. Zhang, J., Tao, D.: Empowering Things with Intelligence: A Survey of the Progress, Challenges, and Opportunities in Artificial Intelligence of Things. IEEE Internet Things J. 8, 7789–7817 (2021). https://doi.org/10.1109/JIOT.2020.3039359.
3. Barreto, L., Amaral, A., Pereira, T.: Industry 4.0 implications in logistics: an overview. Procedia Manuf. 13, 1245–1252 (2017). https://doi.org/10.1016/j.promfg.2017.09.045.
4. Ashton, K.: That "Internet of Things" Thing. 1.
5. Group, S.M.A., Engineering (NITIE), N.I. of I., Lake, V., Mumbai, Group, I.R.A., Engineering (NITIE), N.I. of I., Lake, V., Mumbai, Group, I.T.A., Engineering (NITIE), N.I. of I., Lake, V., Mumbai, India: Internet of Things (IoT): A Literature Review. J. Comput. Commun. 03, 164 (2015). https://doi.org/10.4236/jcc.2015.35021.
6. Silva, B.N., Khan, M., Han, K.: Internet of Things: A Comprehensive Review of Enabling Technologies, Architecture, and Challenges. IETE Tech. Rev. 35, 205–220 (2018). https://doi.org/10.1080/02564602.2016.1276416.
7. Gubbi, J., Buyya, R., Marusic, S., Palaniswami, M.: Internet of Things (IoT): A vision, architectural elements, and future directions. Future Gener. Comput. Syst. 29, 1645–1660 (2013). https://doi.org/10.1016/j.future.2013.01.010.
8. Louis, J., Dunston, P.S.: Integrating IoT into operational workflows for real-time and automated decision-making in repetitive construction operations. Autom. Constr. 94, 317–327 (2018). https://doi.org/10.1016/j.autcon.2018.07.005.
9. Al-Hadhrami, Y., Hussain, F.K.: Real time dataset generation framework for intrusion detection systems in IoT. Future Gener. Comput. Syst. 108, 414–423 (2020). https://doi.org/10.1016/j.future.2020.02.051.
10. Borgohain, T., Kumar, U., Sanyal, S.: Survey of Security and Privacy Issues of Internet of Things. ArXiv150102211 Cs. (2015).
11. Ferrag, M.A., Maglaras, L., Moschoyiannis, S., Janicke, H.: Deep learning for cyber security intrusion detection: Approaches, datasets, and comparative study. J. Inf. Secur. Appl. 50, 102419 (2020). https://doi.org/10.1016/j.jisa.2019.102419.
12. Eskandari, M., Janjua, Z.H., Vecchio, M., Antonelli, F.: Passban IDS: An Intelligent Anomaly-Based Intrusion Detection System for IoT Edge Devices. IEEE Internet Things J. 7, 6882–6897 (2020). https://doi.org/10.1109/JIOT.2020.2970501.
13. Chaabouni, N., Mosbah, M., Zemmari, A., Sauvignac, C., Faruki, P.: Network Intrusion Detection for IoT Security Based on Learning Techniques. IEEE Commun. Surv. Tutor. 21, 2671–2701 (2019). https://doi.org/10.1109/COMST.2019.2896380.
14. Özgür, A., Erdem, H.: A review of KDD99 dataset usage in intrusion detection and machine learning between 2010 and 2015. PeerJ Inc. (2016). https://doi.org/10.7287/peerj.preprints.1954v1.
15. Jan, S.U., Ahmed, S., Shakhov, V., Koo, I.: Toward a Lightweight Intrusion Detection System for the Internet of Things. IEEE Access. 7, 42450–42471 (2019). https://doi.org/10.1109/ACCESS.2019.2907965.
16. Hsu, C.-W., Chang, C.-C., Lin, C.-J.: A Practical Guide to Support Vector Classification. 16.
17. Xu, C., Shen, J., Du, X., Zhang, F.: An Intrusion Detection System Using a Deep Neural Network With Gated Recurrent Units. IEEE Access. 6, 48697–48707 (2018). https://doi.org/10.1109/ACCESS.2018.2867564.
18. Yin, C., Zhu, Y., Fei, J., He, X.: A Deep Learning Approach for Intrusion Detection Using Recurrent Neural Networks. IEEE Access. 5, 21954–21961 (2017). https://doi.org/10.1109/ACCESS.2017.2762418.
19. Li, Z., Qin, Z., Huang, K., Yang, X., Ye, S.: Intrusion Detection Using Convolutional Neural Networks for Representation Learning. In: Liu, D., Xie, S., Li, Y., Zhao, D., and El-Alfy,

E.-S.M. (eds.) Neural Information Processing. pp. 858–866. Springer International Publishing, Cham (2017). https://doi.org/10.1007/978-3-319-70139-4_87.
20. Vinayakumar, R., Soman, K.P., Poornachandran, P.: Applying convolutional neural network for network intrusion detection. In: 2017 International Conference on Advances in Computing, Communications and Informatics (ICACCI). pp. 1222–1228 (2017). https://doi.org/10.1109/ICACCI.2017.8126009.
21. Collective Anomaly Detection Based on Long Short-Term Memory Recurrent Neural Networks | SpringerLink, https://link.springer.com/chapter/10.1007/978-3-319-48057-2_9?utm_source=getftr&utm_medium=getftr&utm_campaign=getftr_pilot, last accessed 2022/03/15.
22. Roy, S.S., Mallik, A., Gulati, R., Obaidat, M.S., Krishna, P.V.: A Deep Learning Based Artificial Neural Network Approach for Intrusion Detection. In: Giri, D., Mohapatra, R.N., Begehr, H., and Obaidat, M.S. (eds.) Mathematics and Computing. pp. 44–53. Springer, Singapore (2017). https://doi.org/10.1007/978-981-10-4642-1_5.
23. Koroniotis, N., Moustafa, N., Sitnikova, E., Turnbull, B.: Towards the development of realistic botnet dataset in the Internet of Things for network forensic analytics: Bot-IoT dataset. Future Gener. Comput. Syst. 100, 779–796 (2019). https://doi.org/10.1016/j.future.2019.05.041.
24. Ashraf, J., Keshk, M., Moustafa, N., Abdel-Basset, M., Khurshid, H., Bakhshi, A.D., Mostafa, R.R.: IoTBoT-IDS: A novel statistical learning-enabled botnet detection framework for protecting networks of smart cities. Sustain. Cities Soc. 72, 103041 (2021). https://doi.org/10.1016/j.scs.2021.103041.
25. Alsaedi, A., Moustafa, N., Tari, Z., Mahmood, A., Anwar, A.: TON_IoT Telemetry Dataset: A New Generation Dataset of IoT and IIoT for Data-Driven Intrusion Detection Systems. IEEE Access. 8, 165130–165150 (2020). https://doi.org/10.1109/ACCESS.2020.3022862.
26. Moustafa, N., Ahmed, M., Ahmed, S.: Data Analytics-Enabled Intrusion Detection: Evaluations of ToN_IoT Linux Datasets. In: 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom). pp. 727–735 (2020). https://doi.org/10.1109/TrustCom50675.2020.00100.
27. Garcia, S., Parmisano, A., Erquiaga, M.J.: IoT-23: A labeled dataset with malicious and benign IoT network traffic, https://zenodo.org/record/4743746, (2020). https://doi.org/10.5281/zenodo.4743746.
28. Abdalgawad, N., Sajun, A., Kaddoura, Y., Zualkernan, I.A., Aloul, F.: Generative Deep Learning to Detect Cyberattacks for the IoT-23 Dataset. IEEE Access. 10, 6430–6441 (2022). https://doi.org/10.1109/ACCESS.2021.3140015.
29. K., G., S.H., B.: Network traffic analysis through deep learning for detection of an army of bots in health IoT network. Int. J. Pervasive Comput. Commun. ahead-of-print, (2022). https://doi.org/10.1108/IJPCC-10-2021-0259.
30. Bolatti, D., Karanik, M., Todt, C., Scappini, R., Gramajo, S.: Intelligent Anomaly Detection System for IoT. In: IX Jornadas de Cloud Computing, Big Data & Emerging Topics. pp. 47–50. Universidad Nacional de La Plata, La Plata (2021).
31. Bolatti, D., Todt, C., Karanik, M., Scappini, R.: Proposed update of Technical Report ITU-T YSTR-IADIoT, "Intelligent Anomaly Detection System for IoT," https://www.itu.int/md/T17-SG020RG.LATAM-C-0014/en, last accessed 2022/04/14.
32. Elsayed, M.S., Le-Khac, N.-A., Jurcut, A.D.: InSDN: A Novel SDN Intrusion Dataset. IEEE Access. 8, 165263–165284 (2020). https://doi.org/10.1109/ACCESS.2020.3022633.
33. Docker Documentation, https://docs.docker.com/, last accessed 2022/04/14.