SVMs applied to DoS Attack Detection

David Carrascal, Adrián Guerrero, Artem Strilets, Pablo Collado

January 2020

Abstract

The purpose of this project is to develop an artificial intelligence to classify possible DDoS attacks in an SDN network. This will be done by using data collectors such as Telegraf, Mininet to emulate the SDN network, and InfluxDB and Grafana as a means to store data and visualize it respectively. For non-English speakers we leave part of the content of this guide written in Spanish:

- Network Scenario - Mininet Guide Link
- DDoS using hping3 tool Guide Link
- Mininet Internals (II) Guide Link

Keywords: *DDoS attacks*; *SDN network*; *Artificial Intelligence classification*; *Mininet*

Contents

1 Notes

Throughout the document we will always be talking about 2 virtual machines (VMs) on which we implement the scenario we are discussing. In order to keep it simple we have called one VM **controller** and the other one **test**. Even though the names may seem kind of random at the moment we promise they're not. Just keep this in mind as you continue reading.

The following document discusses the project version we have uploaded to GitHub. In order to simplify deployment we have decided to automatize the installation of every needed component by means of `bash` scripts. Our intention was to use Ansible but as it's not embedded in Vagrant we prefered not to make people wanting to use our work install yet another tool. As we have studied configuration management in our subject we'll discuss the playbooks used for deployment when presenting our work to the class bit it will **NOT** be talked about here. As the explanation is pretty similar to the one given in the last lab practice and the playbooks are essentially those written by *Pablo Collado* we believe his lab report is enough to slah any doubts the user of the playbooks may come across. In order not to make this document longer than it already is we haven't included it for simplicity. Had it be required we would be pleased to hand it in to anyone interested.

2 Installation Methods

We have created a **Vagrantfile** through which we provide each machine with the necessary scripts to install and configure the scenario. By working in a virtualized environment we make sure we all have the exact same configuration so that tracing and fixing errors becomes much easier. If you do not want to use Vagrant as a provider you can follow the native installation method we present below.

2.1 Vagrant

First of all, clone the repository from GitHub

```
git clone https://github.com/GAR-Project/project
cd project
```

We power up the virtual machine through **Vagrant**:

```
vagrant up
```

And we have to connect to both machines. **Vagrant** provides a wrapper for the *SSH* utility that makes it a breeze to get into each virtual machine. The syntax is just `vagrant ssh <machine_name>` where the `<machine_name>` is given in the **Vagrantfile** (take a look at the appendix for more details):

```
vagrant ssh test
vagrant ssh controller
```

We should already have all the machines configured with all the necessary tools to bring our network up with Mininet on the **test** VM, and Ryu on the **controller** VM. This includes every `python3` dependency as well as any needed packages.

2.1.1 Troubleshooting Problems Regarding `SSH`

If you have problems connecting via SSH to the machine, check that the keys in the path `.vagrant/machines/test/virtualbox/` are owned by the user, and have read-only permissions for the owner of the key.

```
cd .vagrant/machines/test/virtualbox/
chmod 400 private_key

# We could also use this instead of
# "chmod 400" (u,g,o -> user, group, others)
# chmod u=r,go= private_key
```

Instead of using vagrant's manager to make the `SSH` connection, we can opt for manually doing it ourselves by passing the path to the private key to SSH. For example:

```
ssh -i .vagrant/machines/test/virtualbox/ \
private_key vagrant@10.0.123.2
```

2.2 Native

This method assumes you already have any VMs up and running with the correct configuration and dependencies installed. Ideally you should have 2 VMs. We will be running **Ryu** (the *SDN* controller) in one of them and we will have **mininet**'s emulated network with running in the other one. Try to use Ubuntu 16.04 (a.k.a **Xenial**) as the VM's distribution to avoid any mistakes we may have not encountered.

First of all clone the repository, just like how the Kaminoans do it and then navigate into it:

```
git clone https://github.com/GAR-Project/project
cd project
```

Manually launch the provisioning scripts in each machine:

```
# To install Mininet, Mininet's dependencies
# and telegraf. Run it on the "mininet" VM
sudo ./util/install_mininet.sh
sudo ./util/install_telegraf.sh

# To install Ryu and
# the Monitoring system (Grafana + InfluxDB).
# Run it on the "controller" VM
sudo ./util/install_ryu.sh
sudo ./util/install_grafana_influxdb.sh
```

3 Our Scenario

Our network scenario is described in the following script src/scenario_basic.py. Mininet makes use of a Python API to give users the ability to automate processes easily, or to develop certain modules at their convenience. For this and many other reasons, Mininet is a highly flexible and powerful tool for network emulation which is widely used by the scientific community.

- For more information about the API, see its manual.

Figure 1 presents us with the *logic* scenario we will be working with. As with many other areas in networking this logic picture doesn't correspond

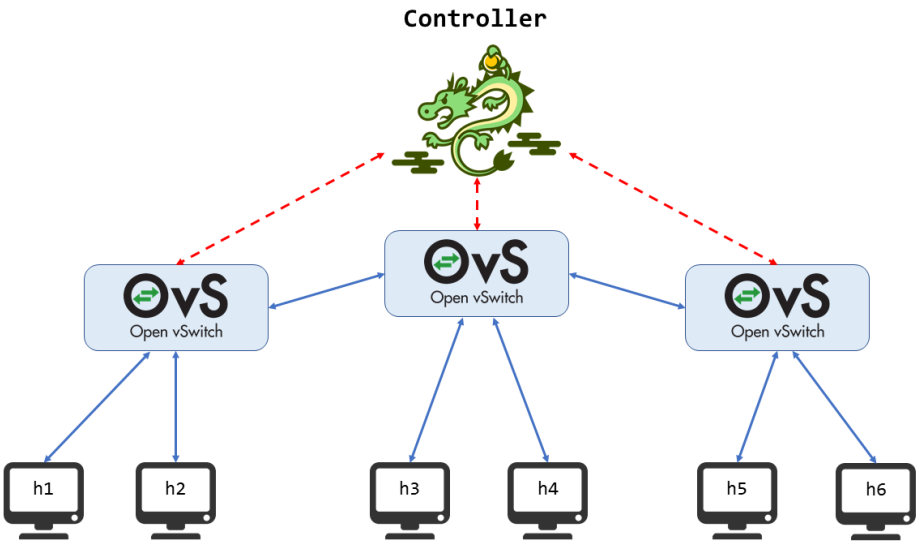


Figure 1: Mininet's Scenario

with the real implementation we are using. We have seen throughout the installation procedure how we are always talking about 2 VMs. If you read carefully you'll see that one VM's "names" are **controller** and **mininet**. So it should come as no surprise that the controller and the network itself are living in different machines!

The first question that may arise is how on Earth can we logically join these 2 together. When working with virtualized environments we will generate a virtual LAN where each VM is able to communicate with one another. Once we stop thinking about programs and abstract the idea of "*process*" we find that we can easily identify the **controller** which is just a **ryu** app, which is nothing more than a **python3** app with the **controller**'s VM **IP** address and the port number where the **ryu** is listening. We shouldn't forget that **any** process running within **any** host in the entire **Internet** can be identified with the host's **IP** address and the processes **port** number. Isn't it amazing?

OK, the above sounds great but... Why should we let the controller live in a machine when we could have everything in a single machine and call it a day? We have our reasons:

- Facilitate teamwork, since the **AI's logic** will go directly into the controller's VM. This let's us increase both working group's independence. One may work on mininet's core and the data collection with **telegraf** whilst the other can look into the DDoS attack detection logic and visualization using **Grafana** and **InfluxDB**.
- Facilitate the storage of data into **InfluxDB** from **telegraf**, as due to the internal workings of Mininet there may be conflicts in the communication of said data. Mininet's basic operation at a low level is be detailed below.
- Having two different environments relying on distinct tools and implementing different functionalities let's us identify and debug problems way faster. We can know what piece of software is causing problems right away!

3.1 Running the scenario

Running the scenario requires having logged into both VMs manually or using vagrant's SSH wrapper. First of all we're going to power up the controller, to do so we run the following from the `controller` VM. It's an application that does a basic forwarding, which is just what we need:

```
ryu-manager ryu.app.simple_switch_13
```

You might prefer to run the controller in the background as it doesn't provide really meaningful information. In order to do so we'll run:

```
ryu-manager ryu.app.simple_switch_13 > /dev/null 2>&1 &
```

Let's break this big boy down:

- `> /dev/null` redirects the `stdout` file descriptor to a file located in `/dev/null`. This is a "special" file in Linux systems that behaves pretty much like a black hole. Anything you write to it just "disappears". This way we get rid of all the bloat caused by the network startup.

- `2>&1` will make the `stderr` file descriptor point where the `stdout` file descriptor is currently pointing (`/dev/null`). Terminal emulators usually have both `stdout` and `stderr`"going into" the terminal itself so we need to redirect these two to be sure we won't see any output.
- `&` makes the process run in the background so that you'll be given a new prompt as soon as you run the command.

If you want to move the controller app back into the foreground so that you can kill it with `CTRL + C` you can run `fg` which will bring the last process sent to the background back to the foreground.

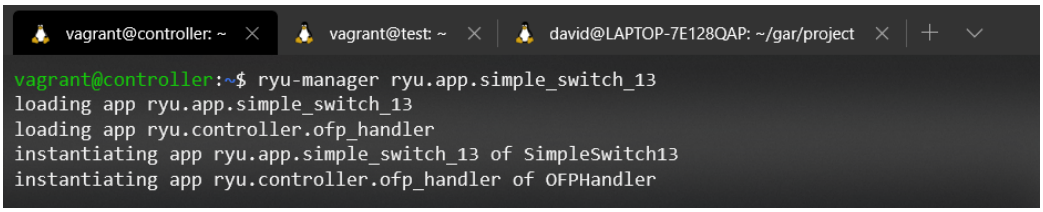


Figure 2: The controller is now running

Once the controller is up we are going to execute the network itself, to do so launch the aforementioned script from the `test` machine:

```
sudo python3 scenario_basic.py
```

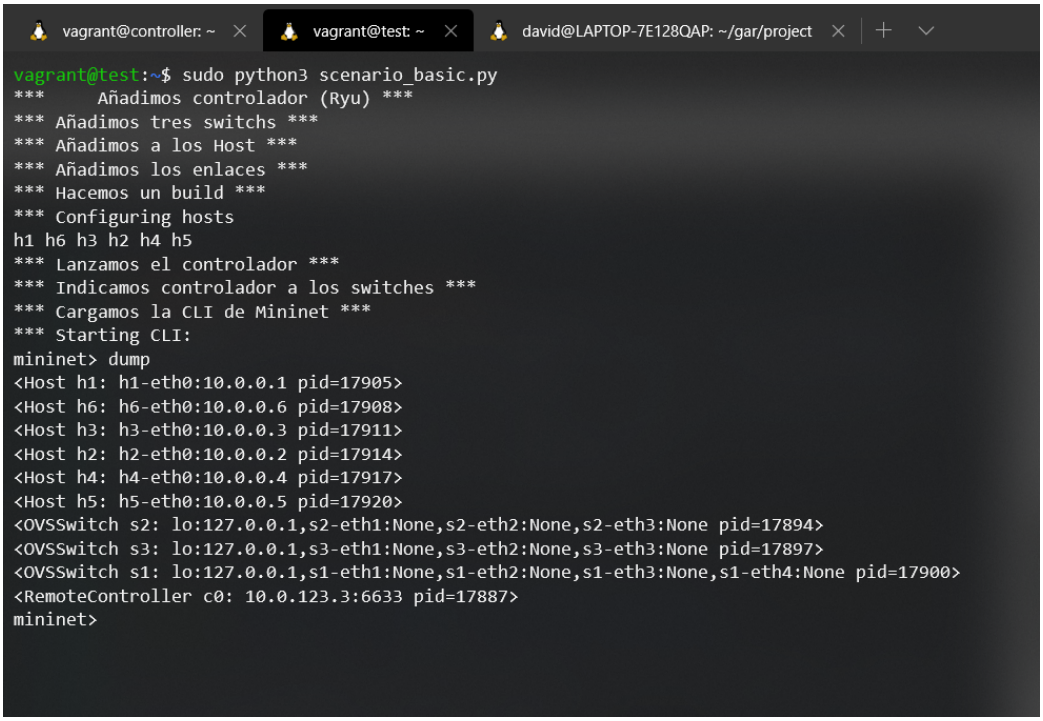


Figure 3: Mininet is now UP

Notice how we have opened **Mininet CLI** from the `test` machine in figure 3. We can perform many actions from this command line interface. The most useful will be detailed below.

3.2 Is it working properly?

We should have our scenario working as intended by now. We can check our network connectivity by pinging the hosts, for example:

```
mininet> h1 ping h3

# We can also ping each other with the pingall command
mininet> pingall
```

As you can see in figure 4 there is full connectivity in our scenario. You may have noticed how the first **ping** takes way longer than the other to get back to use. That is, its **RTT** (**R**ound **T**rip **T**ime) is abnormally high. This

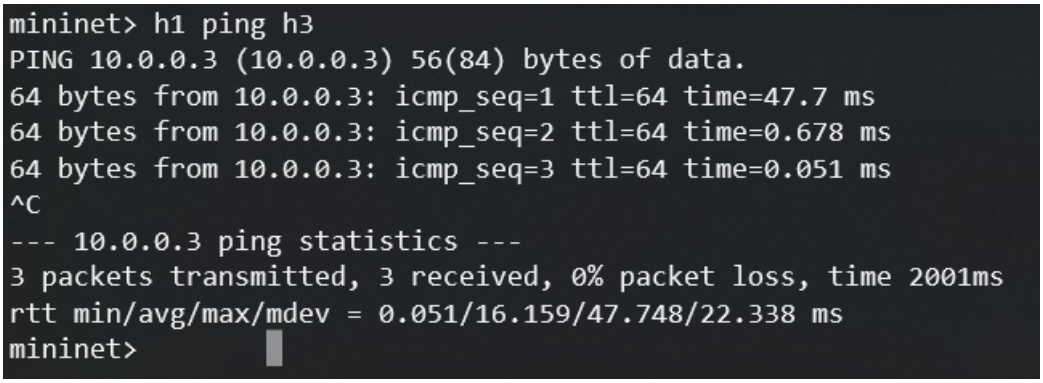


Figure 4: Mininet is working OK

is due to the empty **ARP** tables we currently have *AND* to the fact that we don't yet have a flow defined to handle **ICMP** traffic. We need to take a few steps to fix this:

- An **ARP** resolution between sender and receiver of the ping takes place so that the sender learns the next hop's **MAC** address.
- In addition, the **ICMP** message (ping-request) will be redirected to the driver (a.k.a controller) to decide what to do with the packet as the switches don't yet have a **flow** to handle this traffic type. This way the controller will, when it receives the packet, instantiate a set of rules on the switches so that the **ICMP** messages are routed from one host to the other.

```
vagrant@controller:~$ ryu-manager ryu.app.simple_switch_13
loading app ryu.app.simple_switch_13
loading app ryu.controller.ofp_handler
instantiating app ryu.app.simple_switch_13 of SimpleSwitch13
instantiating app ryu.controller.ofp_handler of OFPHandler
packet in 3 12:1d:25:e5:65:0e 33:33:ff:e5:65:0e 3
packet in 1 12:1d:25:e5:65:0e 33:33:ff:e5:65:0e 4
packet in 2 12:1d:25:e5:65:0e 33:33:ff:e5:65:0e 1
packet in 1 ce:93:36:ee:3c:10 33:33:ff:ee:3c:10 3
packet in 3 ce:93:36:ee:3c:10 33:33:ff:ee:3c:10 1
packet in 1 3a:3d:31:a4:2f:6b 33:33:ff:a4:2f:6b 1
packet in 2 3a:3d:31:a4:2f:6b 33:33:ff:a4:2f:6b 1
packet in 3 3a:3d:31:a4:2f:6b 33:33:ff:a4:2f:6b 1
packet in 3 2e:18:08:fa:36:74 33:33:00:00:00:16 1
packet in 1 3a:3d:31:a4:2f:6b 33:33:00:00:00:16 1
packet in 3 3a:3d:31:a4:2f:6b 33:33:00:00:00:16 1
packet in 2 3a:3d:31:a4:2f:6b 33:33:00:00:00:16 1
packet in 1 ce:93:36:ee:3c:10 33:33:00:00:00:16 3
packet in 2 06:e7:1b:26:d1:07 33:33:00:00:00:16 1
packet in 3 ce:93:36:ee:3c:10 33:33:00:00:00:16 1
```

Figure 5: Ryu is configuring the flows

As you can see in figure 5, the controller's **stdout** (please take a look at the appendix to learn more about file descriptors) indicates the commands it has been instantiating according to the packets it has processed. In the end, for the first packet we will have to tolerate a delay due to **ARP** resolution and **flow** lookup and instantiation within the controller. The good thing is the rest of the packets will already have the destination **MAC** and the rules will already instantiated in the intermediate switches, so the new delay will be minimal.

4 Attack Time

We have already talked about how to set up our scenario but we haven't got into breaking things (i.e the fun stuff). Our goal is to simulate a **DoS** (**D**enial **o**f **Service**) attack. Note that we usually refer to this kind of threats as **DDoS** attacks where the first **D** stands for **D**istributed. This second "name" implies that we have multiple machines trying to flood our own. We are going to launch the needed amounts of traffic from a single host so we would be making a mistake if we were talking about a distributed attack. All in all this is just a minor nitpick, the concept behind both attacks is exactly the same.

We need to flood the network with traffic, great but... How should we do it? We already introduced the tool we are going to be using .

The main objective is being able to classify the traffic in the network as a normal or an abnormal situation with the help of AI algorithms. For these algorithms to be effective we need some training samples so that they can "learn" how to regard and classify said traffic. That's why we need a second tool capable of generating "normal" ICMP traffic so that we have something to compare against. Good ol' **ping** is our pal here.

4.1 Time To Limit The Links

We should no mention our scenario again. We had a **Ryu** controller, three **OVS** switches and several hosts "hanging" from these switches. The question is: **what's the capacity of the network links?**

According to Mininet's wiki that capacity is not limited in the sense that the network will be able to handle as much traffic as the hardware emulating it can. This implies that the more powerful the machine, the larger the link capacity will be. This poses a problem to our experiment as we want it to be reproducible in any host. That's why we have decided to limit each link's bandwidth during the network setup.

This behaviour is a consequence of Mininet's implementation. We'll discuss it when analyzing mininet's internals later on later down the road but the key aspect is that we cannot neglect Mininet's implementation when making design choices!

4.1.1 How To Limit Them

In order to limit the available **BW** (**B**and **W**idth) we'll use Mininet's API. This API is just a wrapper for a **TC** (**T**raffic **C**ontroller) who is in charge of modifying the kernel's **planner** (i.e *Network Scheduler*). The code where we leverage the above is:

```
net = Mininet(topo = None,
                        build = False,
                        host = CPULimitedHost,
                        link = TCLink,
                        ipBase = '10.0.0.0/8')
```

Note how we need to limit each host's capacity by means of the CPU which is what we do through the `host` parameter in Mininet's constructor. We'll also need links with a `TCLink` type. We can achieve this thanks to the `link` parameter. This will let us impose the limits to the network capacity ourselves instead of depending on the host's machines capabilities.

After fiddling with the overall constructor we also need to take care when defining the network links. We can find the following lines over at **src/scenario_basic.py**:

```
net.addLink(s1, h1, bw = 10)
net.addLink(s1, h2, bw = 10)
net.addLink(s1, s2, bw = 5, max_queue_size = 500)
net.addLink(s3, s2, bw = 5, max_queue_size = 500)
net.addLink(s2, h3, bw = 10)
net.addLink(s2, h4, bw = 10)
net.addLink(s3, h5, bw = 10)
net.addLink(s3, h6, bw = 10)
```

We are fixing a **BW** for the links with the `bw` parameter. We have also chosen to assign a finite buffer size to the middle switches in an effort to get as close to reality as we possibly can. If the `max_queue_size` parameter hadn't been defined we would be working with "infinite" buffers at each switch's exit ports. Having these finite buffers will in fact introduce a damping effect in our tests as once you fill them up you can't push any more data through: the output queues are absolutely full... In a real-life scenario we would suffer huge packet losses at the switches and that could be used as a symptom as well but we haven't taken it into account for the sake of simplicity.

We fixed the queue lengths so that they were coherent with standard values. We decided to use a **500 packet** size because *Cisco*'s queue lengths range from 64 packets to about 1000 as found here. We felt like 500 was an appropriate value in the middle ground. With all these restrictions our scenario would look like figure 6.

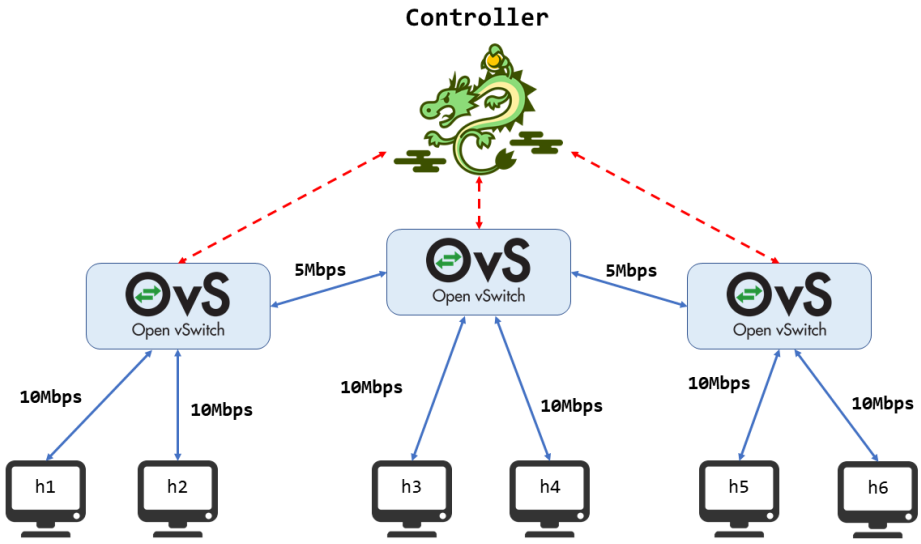


Figure 6: Scenario with link capacities

By inspecting the network dimensions we can see how we have a clear bottleneck... This "flaw" has been introduced on purpose as we want to clearly differentiate regular traffic from the one we experience when under attack.

4.2 Getting Used to `hping3`

This versatile tool can be configured so that it can explore a given network, perform traceroutes, send pings or carry out out flood attacks on different network layers. All in all, it lets us craft our own packets and send them to different destinations at some given rates. You can even forge the source **IP** address to go full stealth mode **ICMP –¿ Echo request (Type = 8, Code = 0)** whilst increasing the rate at which we send them. This will in turn make the network core collapse making our attack successful.

Check out this site for more info on this awesome tool.

4.3 Installing Things... Again!

The tool will be already present on the test machine as it was included in the **Vagrantfile** as part of the VM's provisioning script. In case you want to manually install it you can just run the command below as **hping3** is usually within the default software sources:

```
sudo apt install hping3
```

4.4 Usage

As we have previously discussed this is quite a complete tool so we will only use one of the many functionalities to keep things simple. The command we'll be using is:

```
hping3 -V -1 -d 1400 --faster <Dest_IP>
```

We are going to break down each of the options:

- `-V`: Show verbose output (i.e show more information)
- `-1`: Generate ICMP packets. They'll be ping requests by default
- `-d 1400`: Add a bogus payload. This is not strictly needed but it'll help us use up the link's BW faster. We have chosen a 1400 B payload so as not to suffer fragmentation at the network layer.

- `--faster`: If we used the `flood` option we overwhelmed the virtualized network...

We would like to point out that `hping3` could have been invoked with the `--flood` option instead of `--faster`. When using `--flood` the machine will generate as many packets as it possibly can. This would be great in a world of rainbows but... The virtual network was quickly overwhelmed by the ICMP messages and packets began to be discarded everywhere. Event though this is technically a **DoS** attack gone right too it obscures the phenomena we are faster so we decided to use `--faster` as the rate it provides suffices for our needs.

4.5 Demo Time!

The attack we are going to carry out comprises hosts **1**, **2** and **4**. We'll launch `hping3` from **Host1** targeting **Host4** and we'll try to ping **Host4** from **Host2**. We will in fact see how this "regular" ping doesn't get through as a consequence of a successful **DoS** attack. Figure 7 depicts the situation.

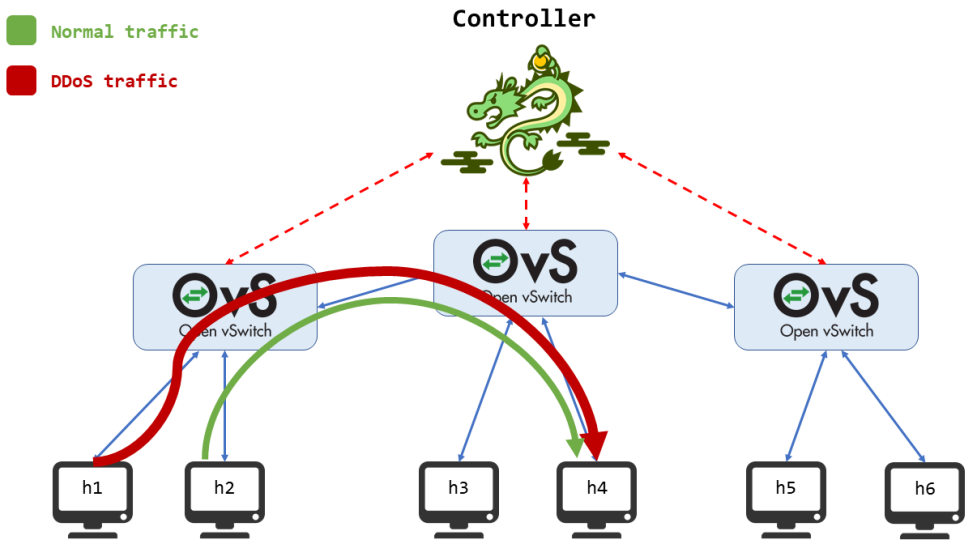


Figure 7: Under attack!

Let's begin by setting up the scenario like we usually do:

```
sudo python3 scenario_basic.py
```

Time to open terminals to both ICMP sources. We'll also fire up `Wireshark` on **Host4** to have a closer look at what's going on. Note the ampersand (`&`) at the end of the second command. It'll detach the `wireshark` process from the terminal so that we can continue running commands as we normally would. To do this we need to run:

```
mininet> xterm h1 h2
mininet> h4 wireshark &
```

Time to launch `hping3` from **Host1** with the parameters we discussed. This is shown in figure 8.

Figure 8: Launching the `DoS` attack

If we now try to ping **Host4** from **Host2** we'll fail horribly as we find in figure 9.

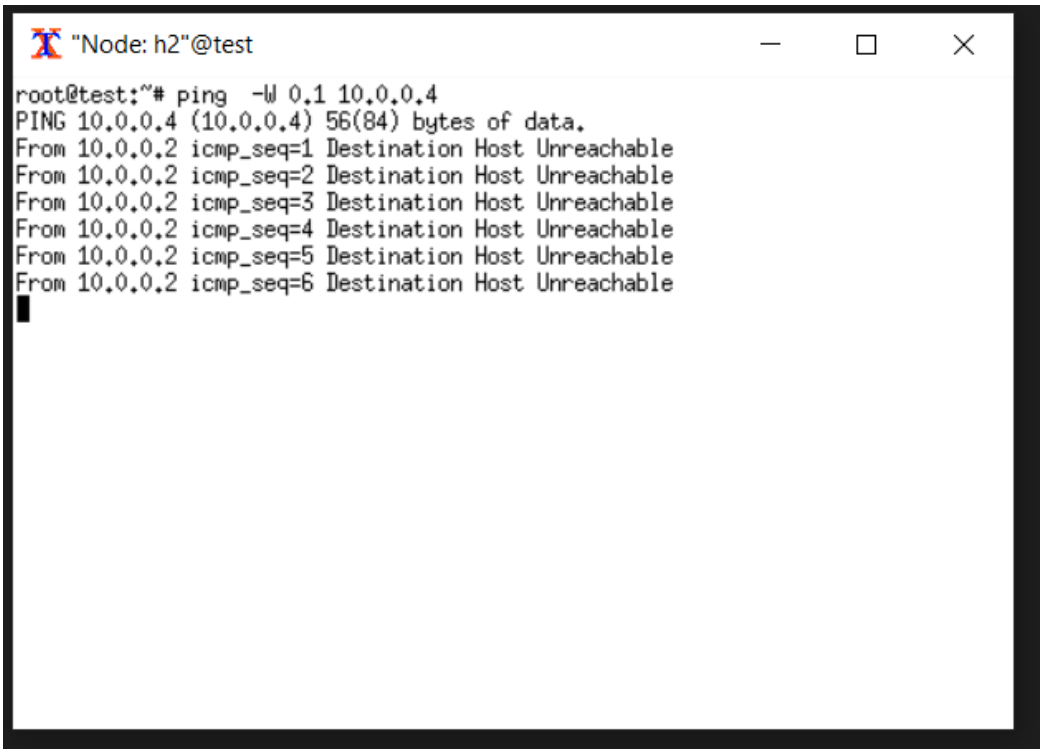


Figure 9: `ICMP` traffic can't get through...

If we halt the **DoS** attack we will see the regular traffic resume its normal operation after a short period of time. Figure 10 depicts this.

We then see how the **DoS** attack against **Host4** has been successful. In order to facilitate issuing the needed commands we have prepared a couple of `python` scripts containing all the needed information so that we only need to run them and be happy. You can find them at:

- Attack `src/ddos.py`
- Regular traffic `src/normal.py`

With all this ready to rock we now need to focus on detecting these attacks and seeing how to possibly mitigate them.

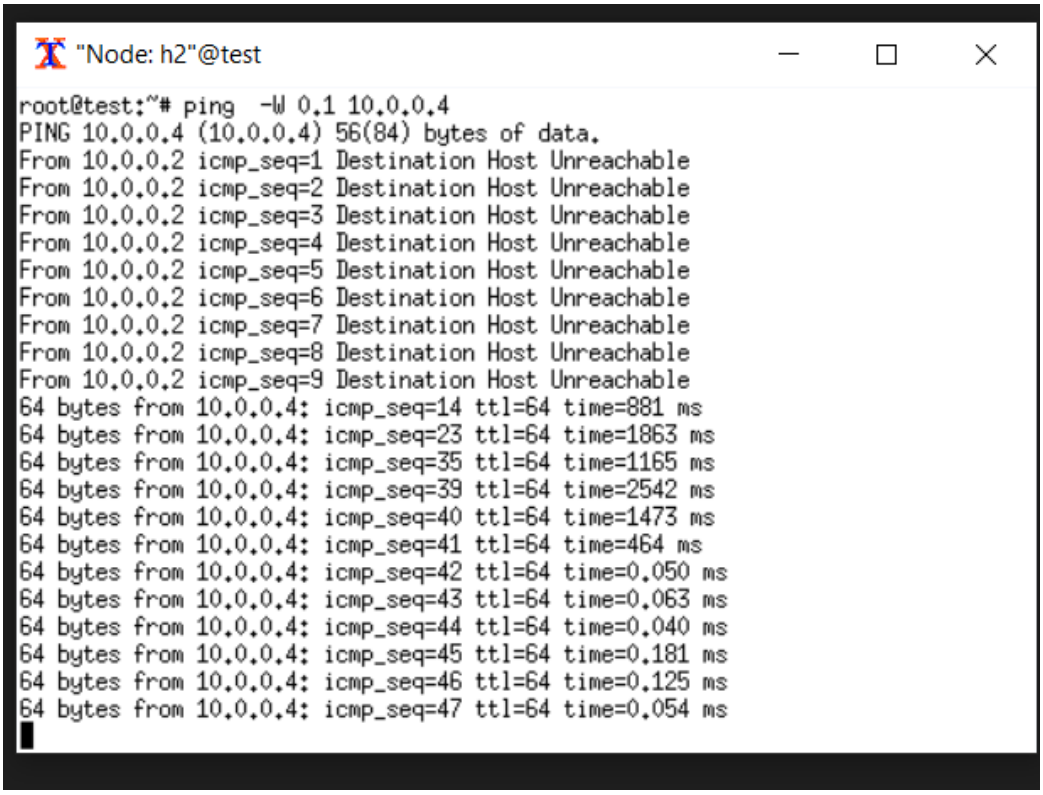


Figure 10: The network has recovered!

4.6 Wanted a Video

You can find a video showing the process we described step by step here. If you stumble upon any questions don't hesitate to contact us!

5 Traffic Classification With a SVM (Support Vector Machine)

We have our scenario working properly and the attack is having the desired effect on our network. In other words, it's blowing things up. If we are to detect the attack we need to gather representative data and process it somehow so that we can predict whether we are under attack or not. As Jack the Ripper once said, let's break this into parts. We'll begin by gathering the necessary data and sending it to a database we can easily query. We'll then prepare training datasets for our SVM and get it ready for making guesses. Let's begin!

5.1 First Step: Getting The Data Collection To Work

5.1.1 What Tools Are We Going To Use?

For a previous project belonging to the same subject we were introduced to both **telegraf** and **influxdb**. The first one is a metrics agent in charge of collecting data about the host it's running on. It's entirely plug-in driven so configuring it is quite a breeze! The latter is a **DBMS** (**D**ata**B**ase **M**anagement **S**ystem) whose architecture is specifically geared towards time series, just what we need! The interconnection between the two is straightforward as one of **telegraf**'s plug-ins provides native support for **influxdb**. We'll have to configure both appropriately and we'll see it wasn't as easy as we once thought due to mininet getting in the way. We have come up both with a "hacky" solution and an alternative any Telecommunications Engineer would be prod of. Just kidding, but it uses networking concepts and not workarounds though.

5.1.2 Leveraging Mininet's Shared Filesystem

Have you ever felt like throwing yourself into `/dev/null` to never come back? That was pretty much our mood when trying to get a host within mininet's network to communicate with the outside world. In order to understand how we ended up "fixing" (it just works) everything we need to go back and take a look at our initial ideas and implementations.

We should not forget that we are looking at `ICMP` traffic in order to make predictions about the state of the network. We first thought about running **telegraf** on a network switch that was directly connected to the controller where our **InfluxDB** instance is running. The good thing about this scheme is that the telegraf process within the switches can communicate with the DB running in the controller through `HTTP`. This is due to the fact that we are invoking the `start()` method of the switches during the network configuration so even though there's no "real" link between them (we didn't create it by calling `addLink()`) they can still communicate.

The above sounds wonderfully well but... switches can only work with information up to the **link layer**, they know nothing about **IP** packets or **ICMP** messages. We should note that **ICMP** is a layer 3-ish (more like layer 3.5) protocol. As it relies on IP for the network services but doesn't have a port number we cannot assign a particular layer to it... All in all the switches knew nothing about ICMP messages crossing them so we find that we need to run telegraf on one of the hosts if we want to get our metrics. In a real case scenario we could devote a router (which can process ICMP data) instead of a switch for this purpose and reconfigure the network accordingly. Anyway we need to get the telegraf instance running in one of the mininet created hosts to communicate with the influx database found in the controller VM. Let's see how we can go about it...

When discussing the internal mechanisms used by mininet later on we'll find out that it relies solely on network namespaces. This implies that the filesystem is shared across the network elements we create with mininet **AND** the host machine itself. This host machine has direct connectivity with the VM hosting the controller so we can take advantage of what others consider to be a flaw in mininet's architecture. We are going to run a telegraf instance on mininet's `Host 4` whose input plug-in will gather ICMP data and whose output will be a file in the VM's home directory. We'll be running a second telegraf instance in the host VM whose input will be the file containing `Host 4`'s output and whose output will be the Influx DB hosted in the controller VM. This architecture leverages the shared filesystem and uses a second telegraf instance as a mere proxy between one of mininet's internal hosts and the controller VM, both living in entirely different networks.

In order to implement this idea we have created all the necessary configuration files under `conf` to then copy them to the appropriate places during Vagrant's provisioning stage.

5.1.3 Implementing A NAT (Network Address Translator) In Mininet For External Communication

Once we implemented the solution above we were able to continue developing the **SVM** as we already had a way of retrieving data. That's why we decided to devote some time to looking for a more elegant solution. Just like we usually do in home LANs we decided to instantiate a NAT process to get interconnection to the network created for the VM's from within the emulated one. Due to problems with the internal functioning of this NAT process provided by Mininet, extra configuration had to be added to achieve the desired connectivity. To solve the problem a series of predefined rules (flows) were installed in each switch to "route" the traffic from our data collector to the NAT process and from there to the outside to InfluxDB. This could be considered a "fix", but in fairness we are only using the logic of an SDN network to route our traffic in the desired way. You can take a closer look at this implementation in this branch.

5.1.4 What Data Are We Going To Use?

We are trying to overwhelm `Host 4` with a bunch (a **VERY BIG** bunch) of `ICMP Echo Requests` (that is fancy for `pings`). By reading through telegraf's input plug-in list we came across the **net** plug-in capable of providing `ICMP` data out of the box.

5.1.5 Getting The Data To `InfluxDB`

Instead of directly sending the output to an influxdb instance we are going to send it to a regular file thanks to the **file** output plug-in. This leads us directly to the configuration of the second telegraf instance.

In this second process we'll be using the **tail** input plug-in. Just like Linux's `tail`, this command will continuously read a file so that it can use it as an input data stream. Instead of polling the file continuously we chose to instead notify telegraf to read it when changes took place. This leads to

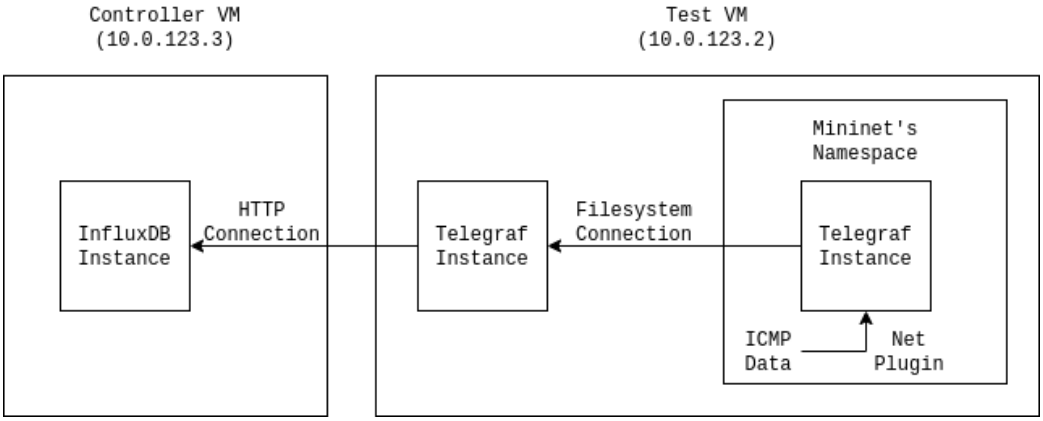


Figure 11: Connection through the filesystem

a more efficient use of system resources overall. The output plug-in we'll be using is now good ol' **influxdb**. We'll point it to the influxdb instance running on the `controller` VM so that everything is correctly connected.

The structure of the system we are dealing with is shown in figure 11.

We are now ready to start querying our database and begin working with the acquired information.

5.1.6 A Note On The Sampling Period

When configuring the interconnection between both telegraf instances we initially left the default `10 s` refresh interval in both. When we read the data we were getting in the DB we noticed some "strange" results in between correct readings so we decided to fiddle with these sampling times in case they were interfering with each other. As we are communicating both processes by means of a file the timing for reading and writing can be critical... We fixed a `2 s` sampling interval in "mininet's" telegraf process and a `4 s` refresh rate in the VM's instance. This means that we are going to get 2 entries in the DB with each update!

After running some tests we found everything was working flawlessly now so we just left it as is.

5.2 Second Step: Generating the Training Datasets

5.2.1 Weren't we using the received `ICMP` messages as the input?

Well... yes and no. The cornerstone for the SVM's input is indeed the number of received ICMP messages **BUT** we decided to use the *derivative* of the incoming packets with respect to time instead of the absolute value. This approach will let the network admin apply the exact same SVM for attack detection even if the traffic increases due to a network enlargement. As we are looking for sudden changes in incoming messages rather than for large numbers this approach is more versatile.

After debating it for a while we settled on including the average of the derivative of the incoming packets as a parameter too. As the mean will vary slowly due to the disparity of the data generated by both situations we'll be more likely to consider the aftermath as an attack too. Even though we may not be subject to very high incoming packet variations any more we'll take a while to resume a normal operation and we decided to let this "recovery time" play a role in the SVM's prediction.

5.2.2 Writting a Script: `src/data_gathering.py`

Once we have the desired data stored in the DB using the SVM becomes a matter of reading it and formatting it so that the SVM "likes it". In order to make the process faster we decided to write a simple python script that uses influxdb's python API to read the data and prepare a **CSV** (**C**omma **S**eparated **V**alues) to later be read by the script implementing the SVM.

The defining quality of training data is meaningfulness. The SVM's predictions will only be as good as the training it received so we need to provide insightful data if we are to get any consistent results.

In order to get appropriate data samples we went ahead an simulated regular traffic by pinging the target host at a rate of roughly 1 ICMP message per second. We then attacked the target until we got around 100 samples into de DB.

Generating the DB is just a matter of reading the DB and outputting the read data to a text file with a `.csv` extension.

5.3 Third Step: Putting it all together

Before analyzing `traffic_classifier.py` let's do a little bit of...

5.3.1 A Tiny Bit of Math

As one of the parameters we feed the **SVM** is the mean of the data we get from the terminal we tried to come up with an scalable way of computing it as more samples came instead of recomputing the mean taking all the values into acount. This naive approach would have needed an ever growing array to store all the data, meaning it would have eventually exhausted the machine's resources... By taking a look at the estimator for the mean ($\hat{x}$) we tried to compute $\Delta\hat{x}$ by taking its previous values into account. Working with the equations yielded:

$$\begin{aligned}
\hat{x}_f &= \hat{x}_o + \Delta\hat{x} \rightarrow \Delta\hat{x} = \hat{x}_f - \hat{x}_o = \\
&= \frac{\alpha+\omega}{\beta+1} - \frac{\alpha}{\beta} = \\
&= \frac{\beta(\alpha+\omega) - \alpha(\beta+1)}{\beta(\beta+1)} = \\
&= \frac{\alpha+\omega-\hat{x}_o\beta-\hat{x}_o}{\beta+1} = \\
&= \frac{\omega-\hat{x}_o}{\beta+1} \rightarrow \\
&\rightarrow \Delta\hat{x} = \frac{\omega-\hat{x}_o}{\beta+1}
\end{aligned} \tag{1}$$

As seen in 1 all we need to do is compute the new mean as $\hat{x}_f = \hat{x}_o + \Delta\hat{x}$ and to compute $\Delta\hat{x}$ we only need the previous mean ($\hat{x}_o$) and values ω, β which are the incoming sample's value and the previous number of samples respectively. This let's us continue to compute the mean indefinitely as long as we store the previous mean and number of recorded samples. You can find this procedure in the `work_time()` method within the `src/traffic_classifier.py` script.

5.3.2 Writing the Script

Apart from the scenario's setup the most important program we wrote is the `traffic_classifier` without a doubt. The file defines the `gar_py()` class which includes a SVM instance, the query used for getting data and many other configuration parameters as its attributes. This let's us use this same technique in other scenarios, in other words, we are increasing this solution's portability.

The class' constructor will limit itself to initializing its attributes and training the SVM by reading the training files we have already prepared. The main thing to note here is how we need to conform to the input format accepted by the SVM itself.

Once it's trained we just need to call the class's `work_time()` method which will enter an infinite loop whose operation can be summarised into these points:

1. Read the last 3 entries in the DB.
2. Verify these entries are indeed new.
3. Update the parameters we're going to use for the prediction.
4. Order the SVM to predict whether the new data represents an attack or not.
5. Write an entry to the appropriate DB signalling whether or not we're under attack.
6. Wait 5 seconds to read new data. New data is sent to the DB every 4 seconds so reading insanely fast is just throwing resources out the window.

Additionally we used `matplotlib` to draw the classification we were carrying out. As you can see in figure 12, the red dots are those data that have been classified as an anomalous traffic, DDoS traffic, and although it seems that there is only one blue dot belonging to "normal" traffic, it is not the case, there are several but their deviation between them is minimal.

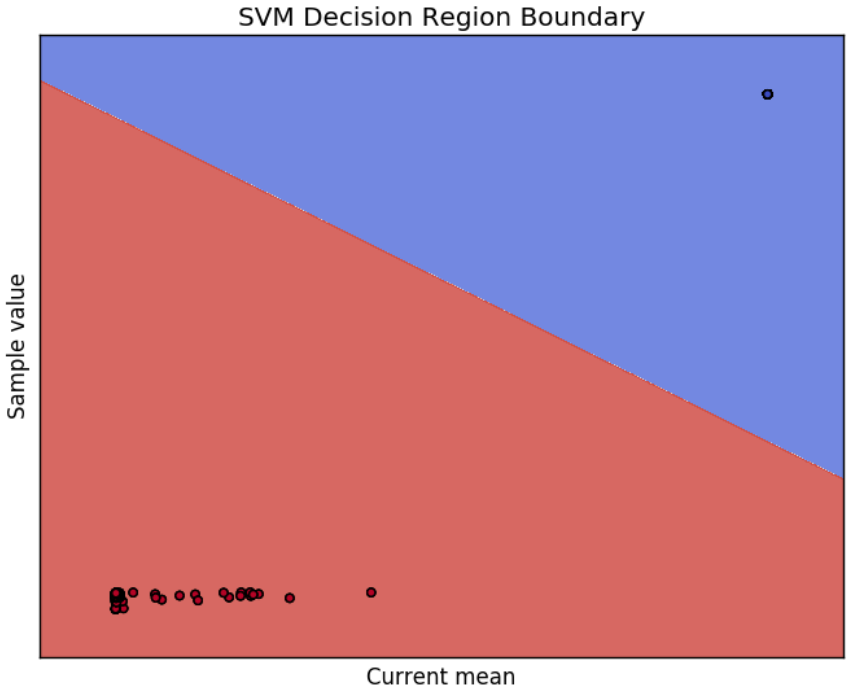


Figure 12: SVM's Decision Regions

We've also written a signal handler to allow for a graceful exit when pressing `CTRL + C`.

And with that we are finished! We hope to have been clear enough but if you still have any questions don't hesitate to contact us. You can find our GitHub at the end of the document.

6 Net Status Visualization with the `Grafana` + `InfluxDB` + `Telegraf` tool set

We have already discussed how to retrieve significant data from the hosts within mininet's network. The thing is this gathered data is nothing more than a collection of numbers whose interpretation can be quite tricky if we don't have a very clear idea of the system we are dealing with. We have seen how the traffic classifier's output was a new measurement within the `h4_net_stats` database within `InfluxDB`. We also talked about the process that let us "sneak" the data past mininet's own network so we are left with the task to interconnect `grafana` and `influxdb` so that we can get the visualization going. Even though we have already stated it, we feel like we should inform the reader that the installation of all these programs is handled by `vagrant`'s provisioning scripts: everything from the download of the correct binaries to the management of the associated service files with `systemd` has been taken care of.

we'll begin by digging a little bit deeper into `grafana` as it's the tool we have described the least. We'll then briefly go through `influxdb`'s and `telegraf`'s data gathering configuration as well. We are doing this for completeness' sake, we should stress once more that the end user doesn't need to deal with.

6.1 Configuring `Grafana`

Grafana is a visualization platform. This implies that `grafana` itself **doesn't** gather any data and it **won't** store it either. This is where `telegraf` and `influxdb` come into play respectively. Having such a modular design allows for changes in the way we gather or store the data whilst maintaining the same monitoring interface. Now, we need to figure out how to get all these three services working cohesively and that's where configuration comes into play. We can take a look at the system we need to handle in figure 13.

The configuration limits itself mostly to editing files under the `/etc` directory in the VMs. Let's dive into `grafana`'s and `influxdb`'s interconnection.

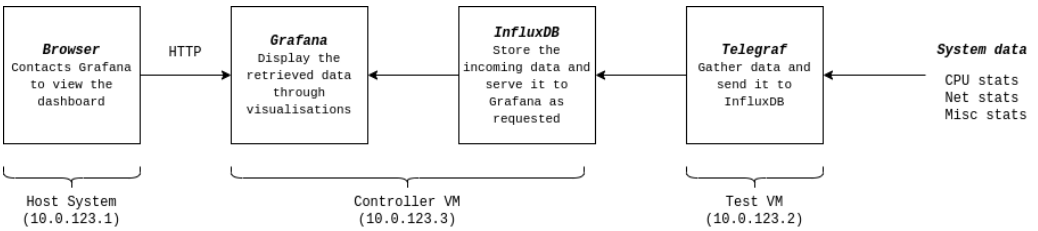


Figure 13: `Grafana`'s, `Influx`'s and `Telegraf`'s connections scheme

6.1.1 Feeding data to `Grafana`

`Grafana` and `influxdb` will communicate through `HTTP` messages. That's why the only "thing" we need to tell `grafana` to reach the database whose data we are interested in is the `IP:port` tuple identifying the machine hosting `inluxdb` and the port it's listening on. As we are running `grafana` in the same machine as the database and we are using `influx`'s default port we just need to use `http://localhost:8086` to reach `influxdb`. We now need to specify the database we want to query. As we have previously discussed we need to take a look at the `h4_net_stats` DB so we'll go ahead and configure that as well.

The above process can be either carried out manually with `grafana`'s GUI or we can opt to provide a `.yml` provisioning file to be used when starting up the service to configure the data sources in a headless way. We opted to g this route to facilitate the deployment of the scenario. You can find the data source configuration files over at `conf/datasources.yaml` in the GitHub repository. This file must be located at `/etc/grafana/provisioning/datasources` in the machine running `grafana`.

6.1.2 Hacker mode on: Preparing the Dashboard

We now have a way of getting data into `grafana` but we haven't provided any way of visualizing it... That's where the dashboard comes in. We'll use `grafana`'s GUI to prepare a set of visualizations ranging from bar graphs to doughnut meters to try and find a coherent and efficient way of determining the network state at a glance. In order to make the dashboard more visually appealing we are also showing measurements related to other aspects of the

Figure 14: `Grafana`'s dashboard

network, not only those regarding the `DoS` attack detection. In order to get these measurements we had to tinker with telegraf's configuration. We'll get into it below. You can find a picture portraying our super cool dashboard in figure 14.

In order to facilitate sharing and reinstalling dashboards `grafana` has a handy export functionality that let's us save our dashboards as `JSON` (**J**ava**S**cript **O**bject **N**otation) files. These can be seamlessly provisioned to `grafana` just like we did with the data source configuration before. You can find the file handling this process over at `conf/main.yaml` which needs to be located at `/etc/grafana/provisioning/dashboards`. The dashboard itself is located at `conf/project_dashboard.json`. This file needs to be located at `/var/lib/grafana/dashboards` so that our provisioning file can find it!

We are now ready to delve deeper into the configuration regarding the database in charge of storing all this data. Enter `InfluxDB`.

6.2 Configuring `InfluxDB`

Getting `influxdb` to work is actually quite a breeze. We can just use the default configuration and take it into account when configuring the other

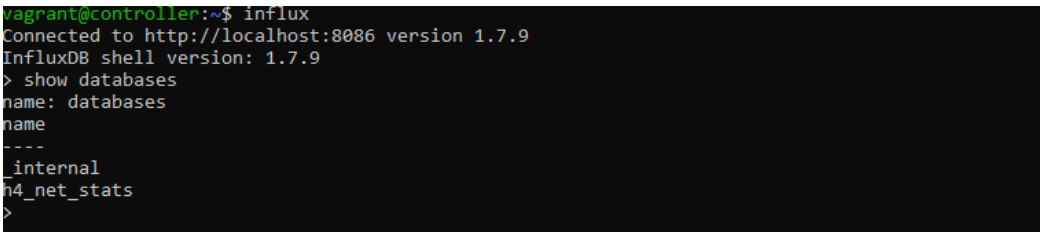


Figure 15: `Influx`'s management system

tools. As you might have guessed, `influxdb` exposes a `HTTP` entry point on port `8086` which is what we used when configuring `grafana` above. You can run `influx` on a shell within the `controller` VM to check that everything is working correctly. The command will open a DB management system where we can select what DB to query and perform operations on. As we haven't started `telegraf` yet we'll see no database has been created... yet. You can find a screenshot showing `influx`'s management system in figure 15.

6.3 The last piece of the puzzle: `telegraf`

`Telegraf`'s configuration is carried out through `.conf` files under `/etc/telegraf` in each machine we want to gather data from. We have already explained the logic behind the filesystem bypass we implemented to "escape" mininet's cage. What we didn't do however is getting into how the "trick" was implemented. In order to get a high level overview of how `telegraf` works we need to talk about `plugins`:

> `Telegraf` is characterized by the implicit modularity of its design. All the data it gathers comes through `input plugins` and all the data it relays or outputs passes through its `output plugins`. We should point out that there are a few "gloabal" configuration parameters for the `telegraf` agent itself; the defaults are acceptable so we can focus solely on the `I/O` plugins. Following this approach we find how we only need to configure an output plugin to send `telegraf`'s output to `influxdb` and we need to choose the appropriate input plugins to gather data of interest for our project. We chose to use the `net` input plugin to access `ICMP` data but there are loads of plugins for gathering from the current memory usage to the total process count in the monitored system. Taking this into account let's us tinker with `telegraf`

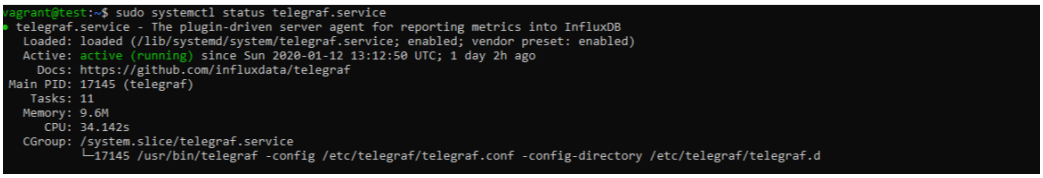


Figure 16: `Telegraf`'s Daemon Status

in a pretty easy and intuitive way whilst abstracting ourselves from the `OS` running "under" us.

Now, as `influxdb`'s `API` is web based we only need to provide the address of the machine hosting it (`localhost` in our case) and we'll be ready to rock. We'll notice how a `telegraf` database will be created seconds after starting `telegraf`'s service. All the data we'll gather is contained within this DB so we only need to configure `grafana`'s dashboard to query it and **magic**, it just works! You can find a screenshot portraying the status of the `telegraf` daemon over at figure 16.

With all these services running we can create awesome monitoring stations with ease. Yay `OpenSource`!

7 Mininet's CLI (Command Line Interface)

We've already set up our scenario and verified that it's working properly. We will now detail the most important commands we can issue from of **Mininet's CLI**.

7.1 Command: `EOF` + `quit` + `exit`

These three commands are used for the same thing, to exit the **Mininet CLI** and finish the emulation. The source code of these three commands does not differ much, **EOF** and **quit** end up using the `do_exit` function at the end, so we could say that they are a bit repetitive. They offer several ways to kill the emulation so that people with different backgrounds feel "at ". The source code taking care of exiting is:

```
def do_exit( self, _line ):
        "Exit"
        assert self # satisfy pylint and allow override
        return 'exited by user command'

def do_quit( self, line ):
        "Exit"
        return self.do_exit( line )

def do_EOF( self, line ):
        "Exit"
        output( '\n' )
        return self.do_exit( line )
```

7.2 Command: `dpctl`

The **dpctl** command is a management utility that allows some control over the OpenFlow switch (ovs-ofctl on the OpenvSwitch). This tool lets us add flows to the flow table, check the features and status of the switches or clean the table among many other things. For example, recall how we previously made a ping between **h1** and **h3**. If we consult the flow tables we will be able to check how the rules for handling **ICMP** flows have been instantiated as seen in figure 17.

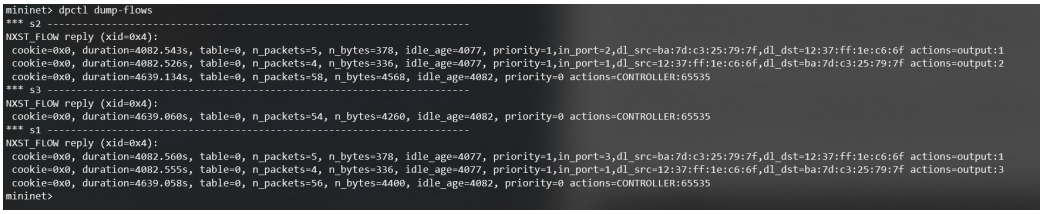


Figure 17: `DumpCTL`

Note how in the first and third switches we have 3 flow instead of the default one that let's us communicate with the controller. On top of that, take a closer look at the third switch and notice how the input and output ports for the first flow are 3 and 1 respectively. The second rule has the exact opposite distribution: the input port is 1 and the output is port 3. This setup let's us establish a communication link through this switch between

any machines hooked to port's 1 and 3. These are the rules the controller has automagically set for us!

This command is quite complex and powerful, and it may not be completely necessary for what we are going to do in this practice. It is nevertheless undoubtedly one of the most important commands to understand the internal workings of **SDN** switches. For more information, we encourage you to take a look at the documentation over at OpenVSwitch.

7.3 Command: `dump` + `net`

These commands will give us information about the emulated topology. The **net** command will indicate the names of the entities in the emulated topology as well as their interfaces. The **dump** command will also indicate the type of entity, its **IP** address, port when applicable, interface and the entity's process identifier (**PID**). You can find a sample output in figure 18.

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=17905>
<Host h6: h6-eth0:10.0.0.6 pid=17908>
<Host h3: h3-eth0:10.0.0.3 pid=17911>
<Host h2: h2-eth0:10.0.0.2 pid=17914>
<Host h4: h4-eth0:10.0.0.4 pid=17917>
<Host h5: h5-eth0:10.0.0.5 pid=17920>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=17894>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=17897>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None,s1-eth3:None,s1-eth4:None pid=17900>
<RemoteController c0: 10.0.123.3:6633 pid=17887>
mininet>
```

Figure 18: `dump's` output

7.4 Command: `xterm` + `gterm`

These two commands will allow us to open terminal emulators in the node identified by the accompanying argument. The command **xterm** will allow us to open a simple **XTERM** (the default terminal emulator for the **X** windows system) terminal emulator, and **gterm** launches a prettier but more resource hungry **Gnome terminal**.

We can open several terminals at once by indicating all the nodes we want to open a terminal in. Later, when we discuss the inner workings of **Mininet**, we'll talk a bit more about where the **bash** process attached to the terminal emulator is running. You might think that this process is totally isolated from the machine on which you are running **Mininet**, but this is not entirely the case... You can find a `xterm` instance running in figure 19.

```
# xterm/gterm [node1] [node2]
xterm h1 h6
```

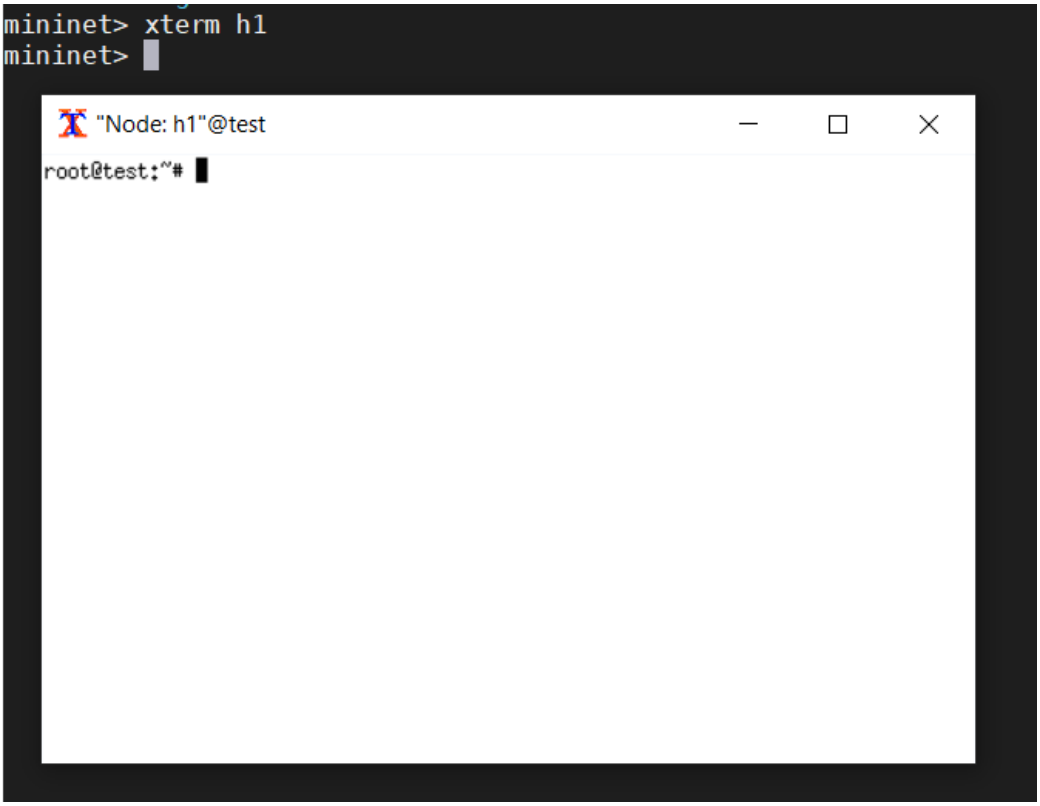


Figure 19: Launching `xterm` on the nodes

7.5 Command: `nodes` + `ports` + `initfs`

These commands will list information related to the nodes in the topology. The **intfs** command will list all information related to the nodes' interfaces. The **nodes** command will show every node in the topology. Finally, the **ports** command is used to list the ports and interfaces of the switches in the topology. An example output can be found in figure 20.

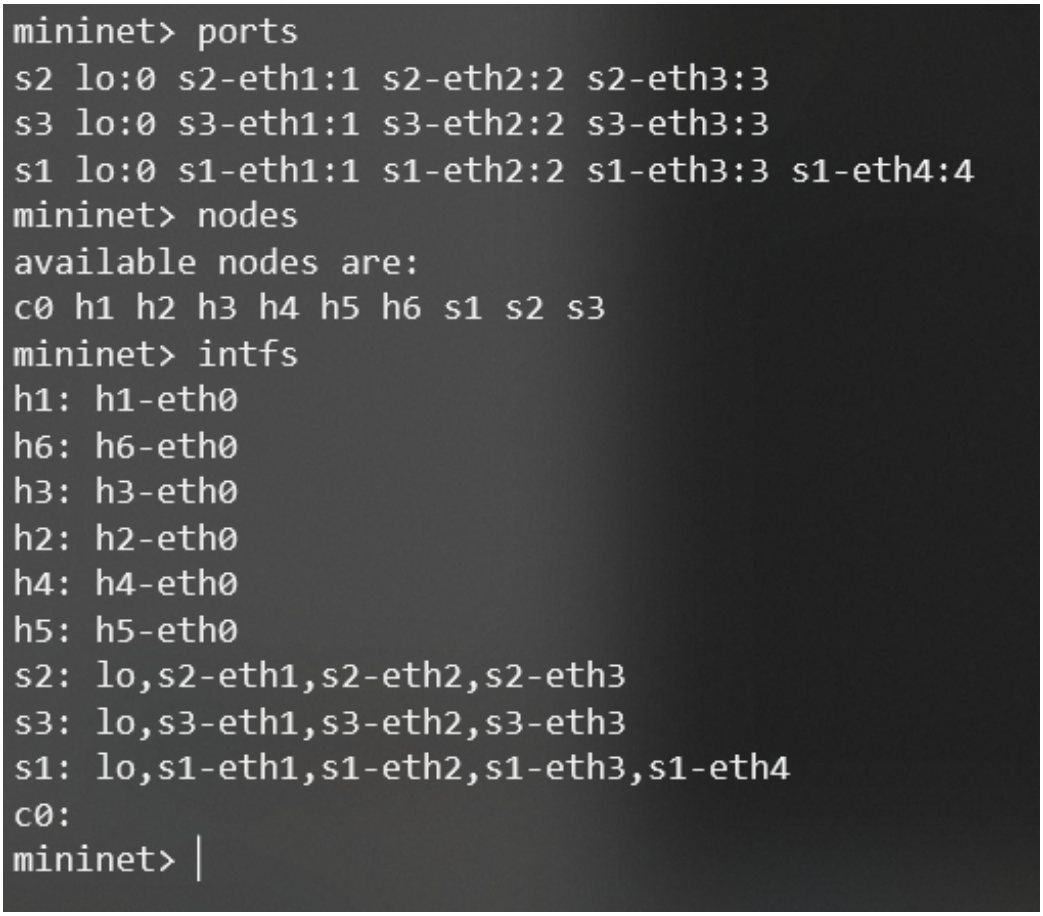


Figure 20: The `ports` and `initfs` commands

7.6 The Rest of the Commands

Someone once told me **manpages** were my friends. This doesn't apply here directly but you get the idea. If you don't know what a command does try running it without arguments and you will be presented with a help section hopefully. If your machine blows up... It wasn't our fault! (It really shouldn't though). You can also issue `help <command_name>` from the **mininet CLI** to gather more intel. You can also contact us directly. We didn't want this section to grow too large and we believe the above commands are more than enough for our purposes.

8 Mininet's Internals

We have been covering **Mininet** for a while now but... What is exactly **Mininet**? It is a tool used for emulating **SDN** (**S**oftware **D**efined **N**etworks). We can write software programs describing the network topology we want and then run them to create a virtual network just like the one we described. Cool right?

Now, notice how we used the term **emulation** instead of **simulation**. Even though many people regard these terms as equivalent they are **NOT** the same. When we talk about **simulation** we are referring to software that computes the outcome of an event given an expected behaviour. On the other hand, **emulation** recreates the scenario under study in its entirety on specific hardware to then study its behaviour.

An example to differentiate the two could be to think about a plane cockpit. If we were to play a video-game like **Flight Simulator** we would be simulating (no surprise) the flight but if we were to practice using a 1:1 scale with real controls we would then be talking about emulation.

Figure 21: A videogame portraying a flight emulator

With this little detail out of the way we could ask ourselves. Does **mininet** emulate or simulate a network?. It is a network **emulator**, here's why. Mininet reserves system resources for each node in the **emulated** network. You might think these nodes are "just" VMs or virtualized containers but... they're not.

That solution would have many advantages but it wouldn't scale to be able to **emulate** large networks or huge amounts of traffic as it would exhaust the host system's resources... The Mininet developers then chose to **exclusively virtualize** what was necessary to carry out the desired **network emulation**.

How did they do it? By using the **Network Namespaces**.

8.1 Network Namespaces

A **network namespace** consists of a logical network stack replica that by default is composed of the **Linux kernel**, paths, **ARP** tables, **Iptables** and network interfaces.

Linux starts with a default **Network namespace** which is the one everyday users need for example. This namespace includes a routing table, an ARP table, the iptables and any network interfaces it might need. The key here is that it is also possible to create more non-default network namespaces. We can then create new devices in those namespaces, or move an existing devices from one namespace to another. This is a rather complex virtualization concept provided by the Linux kernel and we will not delve any further. It is quite interesting if you ask us though...

In this way, each network element has its own network namespace, i.e. each element has its own network stack and interfaces. So at the networking level, one could say, they are independent elements. The key is that every node shares the same process namespace, IPCs namespace, filesystem... We are virtualizing up to the network layer only. This is the true power of the

network stack approach to things. As Vegeta would put it: "The network namespace's power is over 9000!" just like we see him say in figure 22.

Figure 22: The namespaces' power is...

In figure 23 we can see how we created a process in the host machine with the `sleep` command whose **PID** is `20483`. If the network elements were really isolated we wouldn't be able to see this process from other machines but the reality is different with mininet as we discussed.

This is something to assume when working with Mininet's low-cost emulation. This approach would be lacking in other scenarios but it is more than enough to emulate a network. This fact casts some doubts on how to integrate our data collection system with **telegraf** in the different network elements without any incompatibilities...

That's why we decided to take the controller "out of" the machine where Mininet was going to run so as to avoid problems with by-passes by IPCs from telegraf to the InfluxDB database. The only thing left for us to do is to figure out how to correctly install and configure telegraf so that everything works as intended.

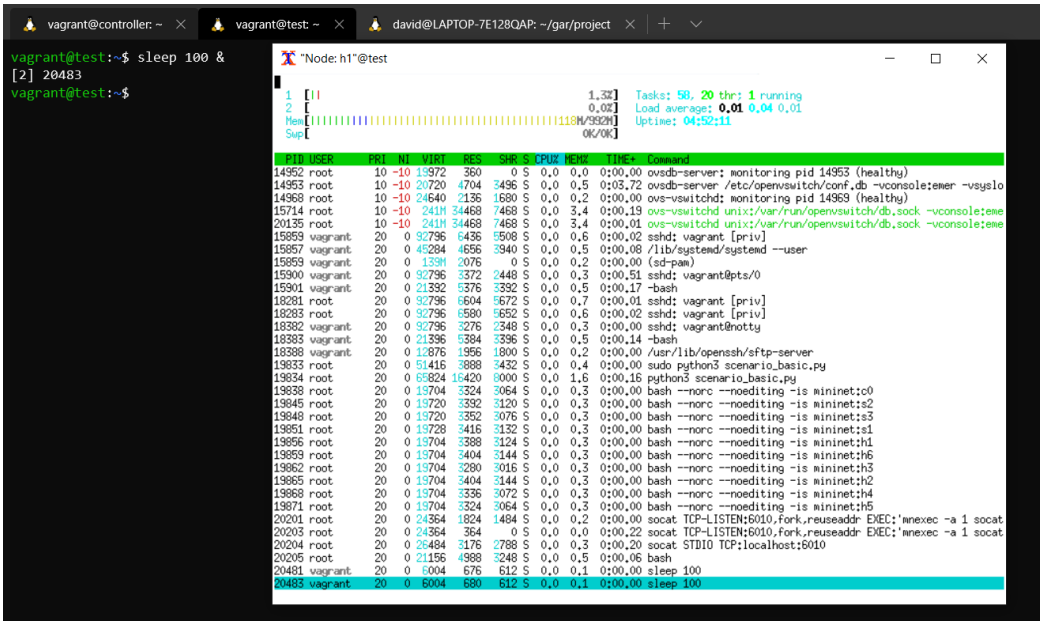


Figure 23: A lazy process...

9 Mininet's Internals (II)

In this second part on the internal operation of Mininet, we will investigate the Kernel-level topology recreated by Mininet to set up our scenario. Finally, we will explain the different ways to raise services in the different Network namespaces, necessary to collect information with `telegraf`.

9.1 Is Mininet Actually Using Namespaces?

We have previously introduced that Mininet makes use of Network namespaces as a method to virtualize network stacks independent of each other, so that we can emulate networks at a minimum cost, but how can we be so sure that it really makes use of them? Here are the steps to verify whether or not Mininet is using Network namespaces.

The first thing we have to do is run the scenario so that mininet can create the network namespaces that it needs to create. In addition, we can also run the controller in the controller machine to check at the last moment that none of the checks made have affected the operation of our scenario.

```
# On the test machine, to set up the topology.
sudo python3 scenario_basic.py

# On the controller machine, to run the controller
ryu-manager ryu.app.simple_switch_13
```

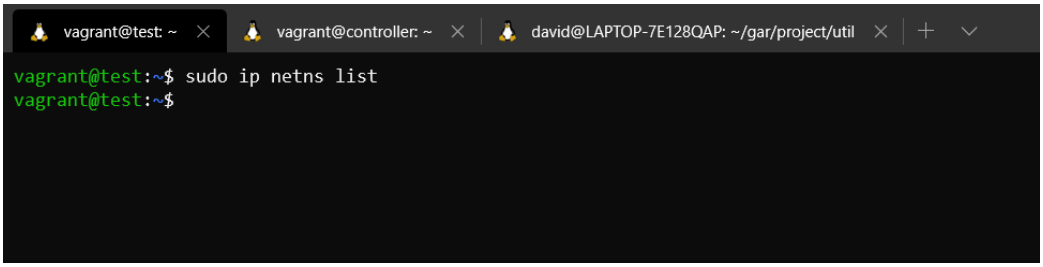
Now that we've set the scenario up we should be able to see if there are any Network namespaces on our machine, to do this we'll use the **iproute2** toolkit. Within this pack we will keep the most famous tool, `ip`. The `ip` tool is becoming established in the new Linux distributions as the de facto tool to work on everything related to Networking in a Linux environment. In the latest versions of Ubuntu for example, the `ifconfig` command is starting to be replaced by the iproute2 toolkit (a.k.a `ip`). This tool has many modules, for more information see its manual:

- Tool's manual: `ip`

The module that we will need to work with Network namespaces is **netns**, we can see all that it can offer us by doing `ip netns help`. The main command to list the Network namespaces using the netns module is the following:

```
sudo ip netns list
```

Knowing the command to list Network Namespaces, and having previously set up the scenario, we'll check if there really are any Network Namespaces created on our machine in figure 9.1.



After checking figure 9.1 it seems that there is no Network namespace created, maybe, **Mininet doesn't work as we said before?** First of all, let's calm down just like **Tux** in figure 9.1, we don't have to rewrite all the documentation.

9.1.1 Not Today!

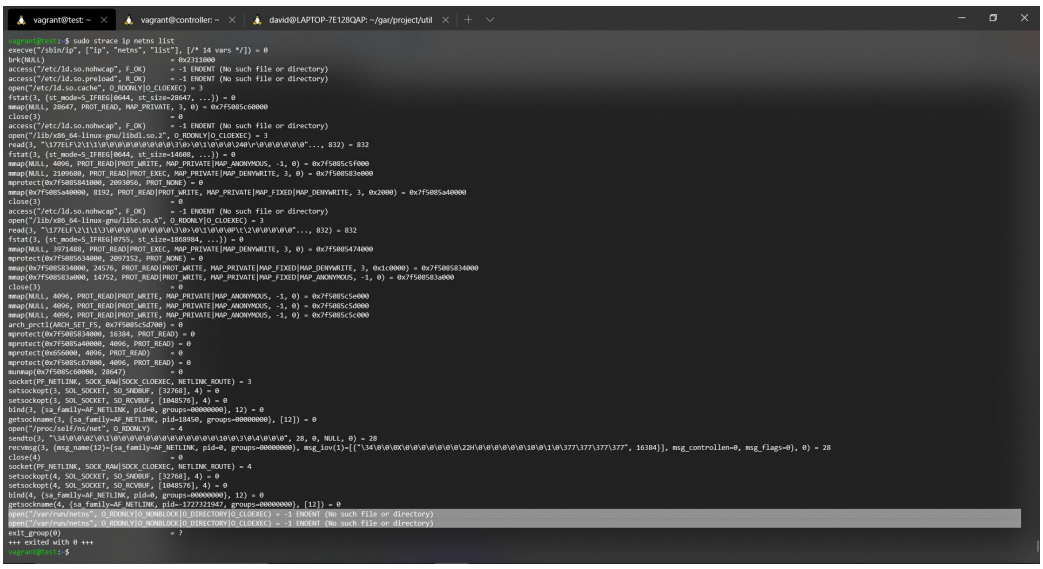
The problem that the command `ip netns list` **doesn't** give us information, is that mininet is not creating the required softlink for the tool to be able to list the network namespaces, if we read the documentation we can find out that `ip netns list` reads from the path `/var/run/netns/` where all the named network namespaces are placed.

If you've gotten to this point you probably want to check that iproute2 really does read from where it says it reads. We can get a trace of the system, i.e. collect all the syscalls made by a program and debug them ourselves. To do this we will use the `strace` command. For more information see their manual.

The command we will use to get the syscalls trace is the following:

```
sudo starce ip netns list
```

Then the output obtained will be like the one in figure 9.1.1 (try to zoom in on the image).



Let's take a good look at the last four lines from figure 9.1.1. If it doesn't look right in the picture, it's these lines:

```
open("/var/run/netns", O_RDONLY|O_NONBLOCK|O_DIRECTORY|
O_CLOEXEC) = -1 ENOENT (No such file or directory)
exit_group(0) = ?
+++ exited with 0 +++
```

The first part of the trace is going to be omitted since the only thing it does is, parse the parameters introduced, load very basic dynamic library functions in Linux (`*.so` files, shared objects, for example, cache, libc among others). We will keep the last lines of the trace where you can see perfectly how it tries to make an `open`, in read mode of the directory, but this **does not exist**.

So we can say that the `ip netns list` command does work correctly. But then, where are the network namespaces used by Mininet?

9.1.2 So where are Mininet's Network Namespaces located?

Well, to answer this question, we must first understand one thing. The `ip` tool with its **netns** module acts as a wrapper when we work with Network namespaces. Namespaces (there are several types) have a finite life, that is, they live as long as they are **referenced**. A namespace can be referenced in **three** ways:

- As long as there's **a process running** inside this namespace.
- As long as you have opened a file descriptor to the namespace file. (`/proc/pid/ns/type_namespace`)
- As long as there is a bind mount of the file (`/proc/pid/ns/namespace type`) of the namespace in question.

If none of these conditions are met, the namespace in question is **deleted**. If it is a `net` type namespace (a.k.a Network Namespace) those interfaces that are in the disappearing namespace will return to the default namespace. Once we understand this concept, we must think about the nature of the Network namespaces that Mininet creates.

Mininet, when is launched it creates an emulated network, when is closed it should disappear, this process should be as light and fast as possible to provide a better user experience. The nature of Mininet's needs leads us to believe that the creation and destruction of network namespaces is associated with the first condition of referencing a namespace. That is, there would be no point in making mounts or softlinks that will have to be removed later, as this would mean a significant workload for large network emulations and an increase in the time spent cleaning up the system once the emulation is complete. In addition, we must take into account that there is a third condition that is quite suitable with Mininet's needs, since only one process is needed running per Network namespace, and when cleaning we must only finish with the processes that *support* the Network namespaces.

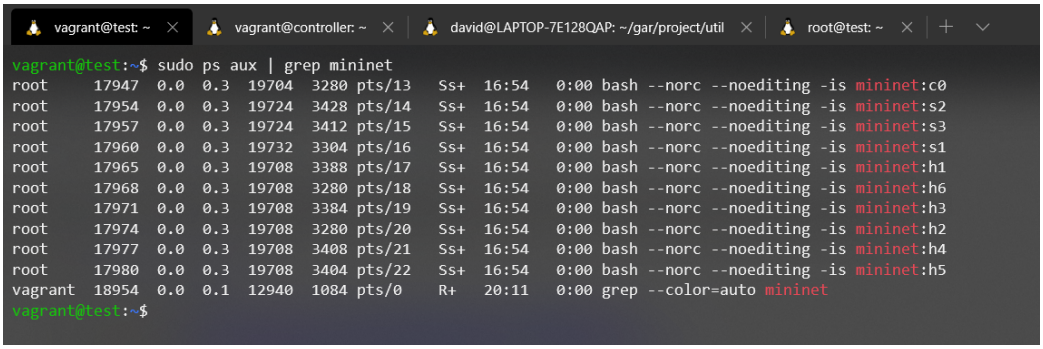
9.1.3 Just a hypothesis?

Well, according to the above reasoning, we should see several processes that are created at the time of the build-up of our scenario in Mininet. These processes should each have a Network Namespace file, `/proc/pid/ns/net`, with a different **inode** for those processes running in different Network namespaces. Where do we start looking?

Let's set the scenario up if we haven't set it up before, list all processes, and filter by the name of *mininet*. Let's see what we find.

```
sudo ps aux | grep mininet
```

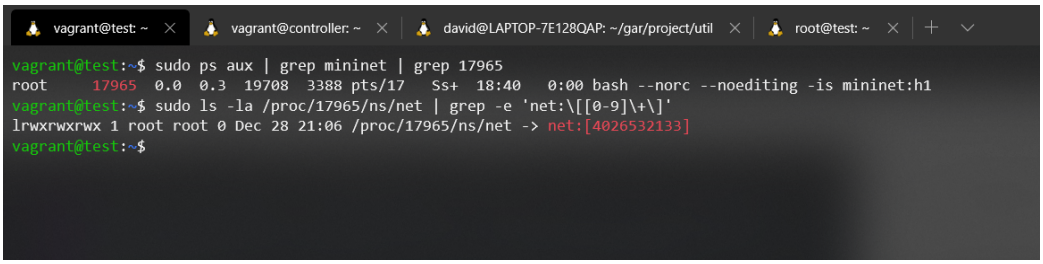
The output of the avobe command can be seen in figure 9.1.3.

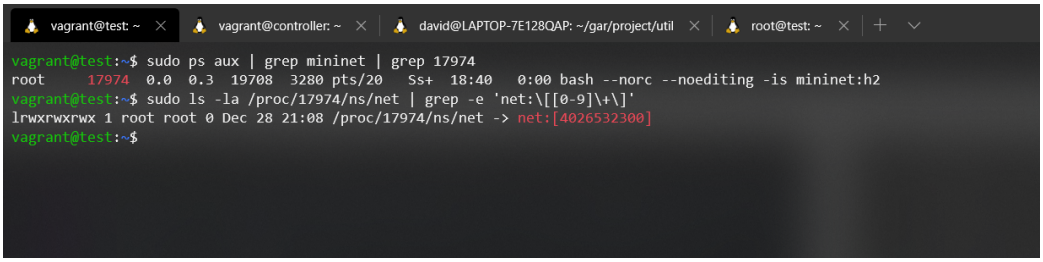


Wow! Without having created any process associated with each node in our scenario, there is already a process running a bash associated with each

element in the scenario at the start of the emulation. That's funny... Isn't it? Let's dig a little deeper.

If we inspect the `/proc/pid/ns/net` file for each process we can see which ones are in a different network namespace depending on the value of the inode. For example, let's check the processes associated to hosts 1 and 2 can be seen in figures 9.1.3 and 9.1.3 respectively.

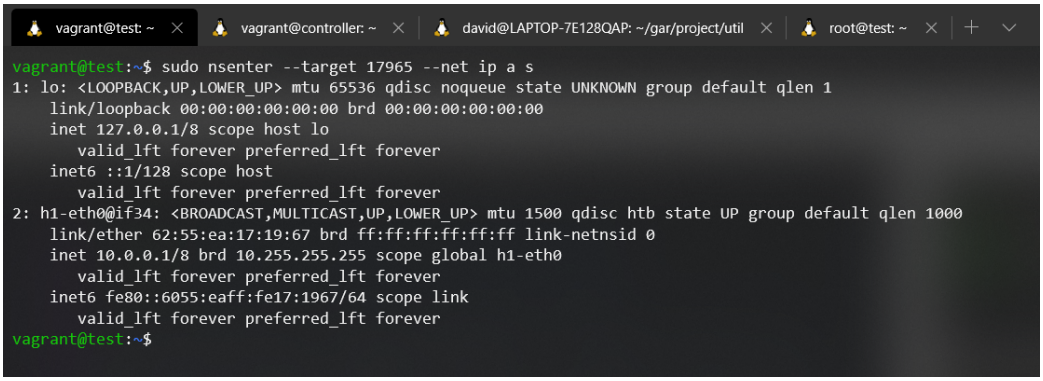


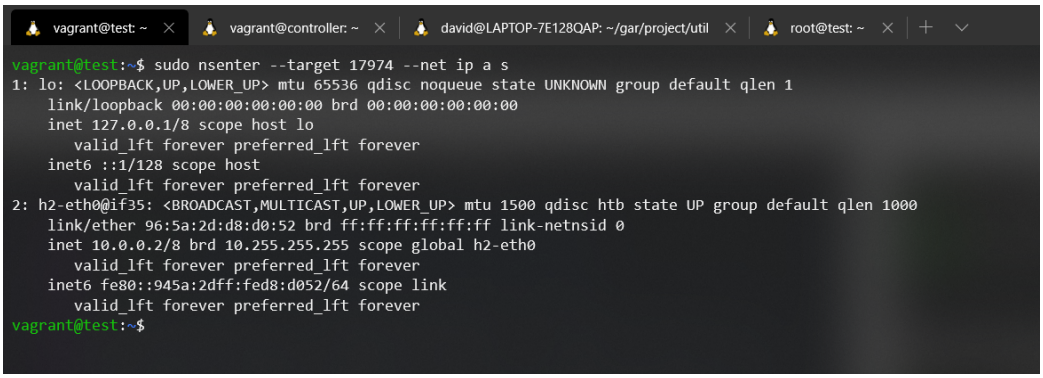


As you can see, different inodes, different files, **different network namespaces**. In order to make it more evident, we are going to execute a command to show which interfaces are associated to each Network namespace. In order to inject processes into a namespace we will use the `nsenter` tool. For more information about this tool, please refer to its manual.

```
nsenter --target <pid> --net {Command}
```

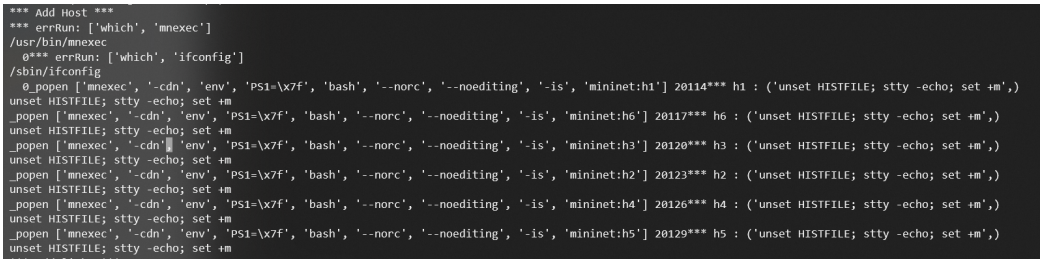
The output of the above command for hosts 1 and 2 can be seen in figures 9.1.3 and 9.1.3 respectively.





If we look at the command entered in each network namespace it is the same, `ip addr show` (a.k.a `ip a s`). With this command we can list all the addresses assigned to each interface of the Network namespace. The result obtained from the execution of each command is the expected one, in the Network namespace of the **Host1** we can see that the interface `h1-eth0` exists, and in the Network namespace of the **Host2** the interface `h2-eth0`. With this test we conclude with the existence of the Network namespace that Mininet uses.

Additionally we can corroborate our hypothesis by changing the "*verbosity*" of our script, where we build the whole scenario topology, `src/scenario_basic.py`, we can change the level of `info` to `debug`, and launch the script again.




```
if __name__ == '__main__':
        #setLogLevel('info')
        setLogLevel('debug')
        scenario_basic()
```

As you can see in the execution output from figure 9.1.3, `veth` is created (**V**virtual **Eth**ernet devices), and the different processes that *will* support the different Network Namespaces. Furthermore, it has been possible to check how the `tc` (**T**raffic **C**controller) is used to establish the bandwidth and maximum queue limits to the links in the scenario.

9.1.4 So, is it possible to use `iproute2` with Mininet?

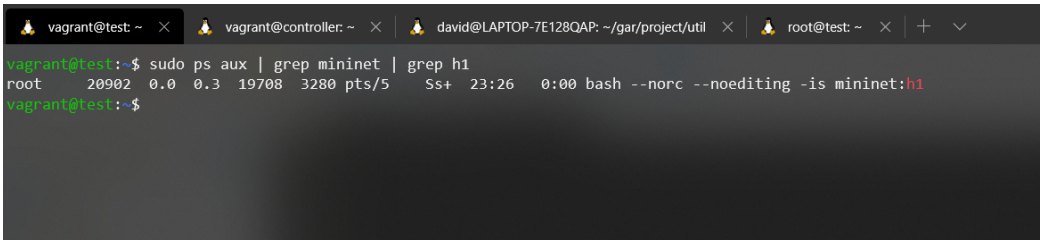
The quick and easy answer in the current state would be that **no**. We can always make use of the Python API to run things inside a network element or if not, we can ultimately open the Mininet CLI, open an xterm and throw things by hand, or as we have done before make use of the `nsenter` tool.

So, there is no solution ? Well, almost everything has a solution, it depends on us to how far we want to go to fix things. Let's see how we can enable the Network namespace of **Host1** to be visible for `ip netns`.

First we must locate the PID of the bash that holds the Host1 Network Namespace. In our case it is the following:

```
sudo ps aux | grep mininet | grep h1
```

The output of the avobe command can be seen in figure 9.1.4.



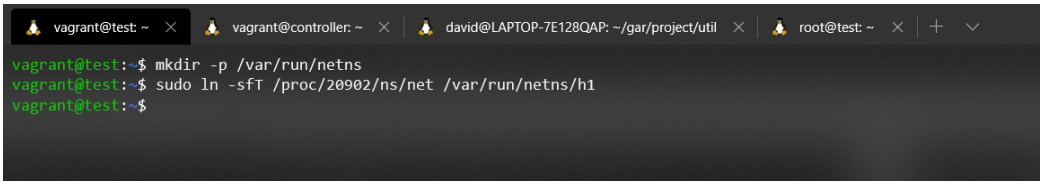
Once we know the PID of the process that *holds* the Host1 Network Namespace, we will create the `/var/run/netns` directory in case it is not created:

```
# We use the -p parameter so that in case it exists
# it does not give us errors.
mkdir -p /var/run/netns/
```

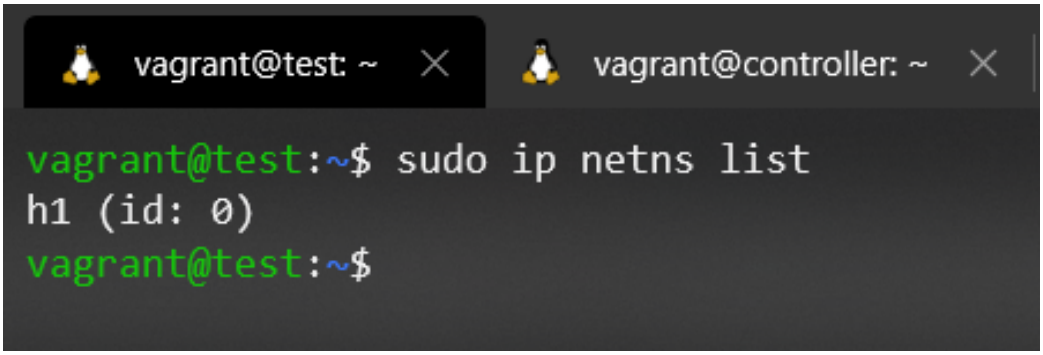
We must make a softlink from the original Network Namespace file in the created directory (Let's remember that this is the path where the `ip netns list` command reads from).

```
sudo ln -sfT /proc/<PID>/ns/net /var/run/netns/h1
```

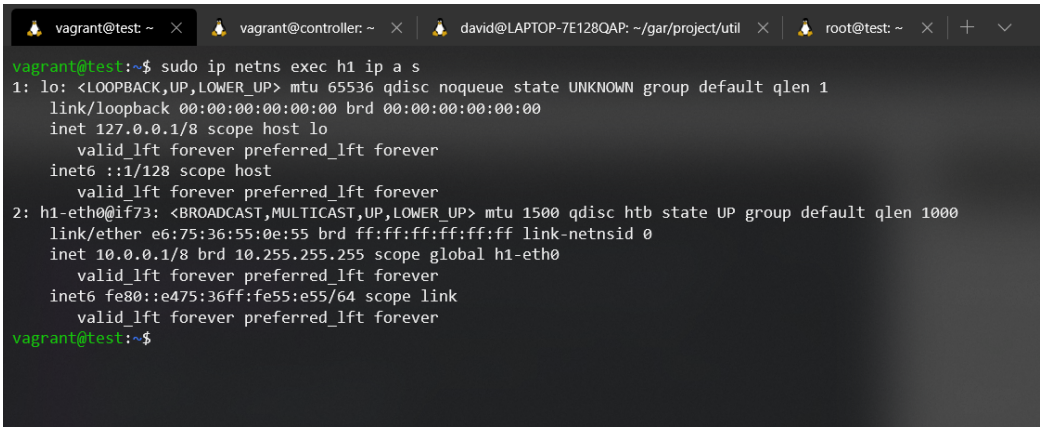
The output of the above command can be seen in figure 9.1.4.



Finally, we would only have to try again the command 'ip netns list' to see if it is able to list the Network namespace. The output can be seen in figure 9.1.4.



Some will say it's dark magic.. But, it's just that, creating a softlink and knowing how each element works.



As you can see in figure 9.1.4, the command is fully functional. You can see how we are able to list all the interfaces of Host1's Network namespace. But as everything, it always has pros and cons, when we make the arrangement of creating a softlink and turning off the emulation with its corresponding system cleaning (we are mainly concerned with the elimination of the processes that supported the Network namespace), we are left with a broken softlink pointing to a site that no longer exists, or is no longer useful.

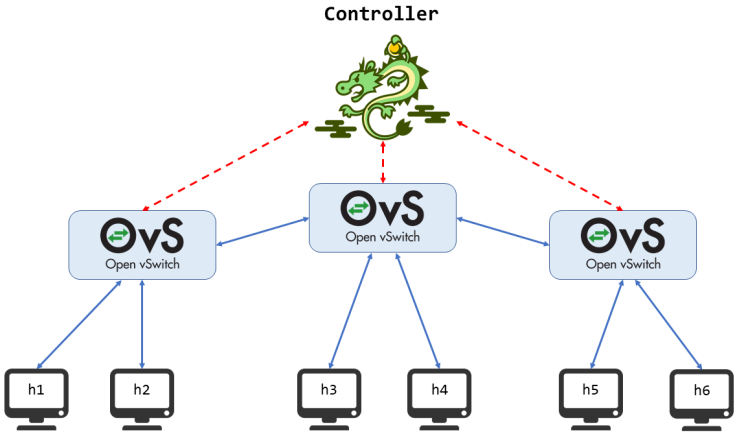
```
*** Stopping 6 hosts
h1 waiting for 20902 to terminate
h6 waiting for 20905 to terminate
h3 waiting for 20908 to terminate
h2 waiting for 20911 to terminate
h4 waiting for 20914 to terminate
h5 waiting for 20917 to terminate

*** Done
vagrant@test:~$
```

```
vagrant@test:~$ sudo ip netns list
h1
vagrant@test:~$ ls -la /var/run/netns/
total 0
drwxr-xr-x  2 root root  60 Dec 28 23:38 .
drwxr-xr-x 25 root root 960 Dec 28 18:30 ..
lrwxrwxrwx  1 root root  18 Dec 28 23:38 h1 -> /proc/20902/ns/net
vagrant@test:~$
```

With the outputs shown in figures 9.1.4 and 9.1.4, it is left up to the user to decide whether or not to use the iproute2 tool. If this is the case, it is recommended that an auxiliary cleaning script be developed to clean up those softlinks that are broken in the `/var/run/netns` directory when the emulation is finished.

9.2 The Big Picture

Once we have concluded that Mininet makes use of Network namespaces and we know how to demonstrate it, we will inspect each of the Network namespaces to draw a scheme of how our Kernel-level scenario is implemented. Let's take another look at our scenario in figure 9.2.

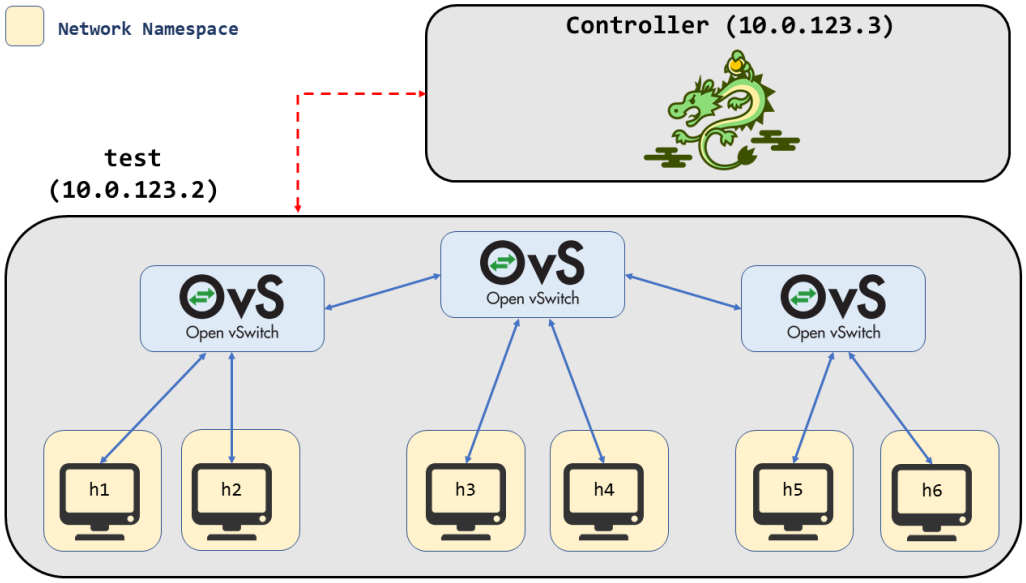


As you can observe in figure 9.2, the switches are network elements that are supposed to be isolated in a network namespace, but for our surprise they are not in the default network namespace. Why does it work then, because there is no by-pass to the default network stack? This is because of the nature of veth, which goes straight to the OVS process itself. (A future guide will attempt to address this issue more fully).

```
vagrant@test: ~ | vagrant@controller: ~ | david@LAPTOP-7E128QAP: ~/gar/project/util | root@test: ~

vagrant@test:~$ sudo ps aux | grep mininet
root     21764  0.0  0.3  19704  3376 pts/1    Ss+  00:02   0:00 bash --norc --noediting -is mininet:c0
root     21771  0.2  0.3  19724  3292 pts/2    Ss+  00:02   0:00 bash --norc --noediting -is mininet:s2
root     21775  0.2  0.3  19724  3424 pts/3    Ss+  00:02   0:00 bash --norc --noediting -is mininet:s3
root     21778  0.2  0.3  19732  3308 pts/4    Ss+  00:02   0:00 bash --norc --noediting -is mininet:s1
root     21783  0.1  0.3  19708  3280 pts/5    Ss+  00:02   0:00 bash --norc --noediting -is mininet:h1
root     21786  0.0  0.3  19708  3408 pts/6    Ss+  00:02   0:00 bash --norc --noediting -is mininet:h6
root     21789  0.0  0.3  19708  3280 pts/7    Ss+  00:02   0:00 bash --norc --noediting -is mininet:h3
root     21792  0.1  0.3  19708  3328 pts/8    Ss+  00:02   0:00 bash --norc --noediting -is mininet:h2
root     21795  0.1  0.3  19708  3280 pts/9    Ss+  00:02   0:00 bash --norc --noediting -is mininet:h4
root     21798  0.1  0.3  19708  3400 pts/10   Ss+  00:02   0:00 bash --norc --noediting -is mininet:h5
vagrant  22276  0.0  0.0  12940   924 pts/0    S+   00:02   0:00 grep --color=auto mininet
vagrant@test:~$ sudo ls -la /proc/21771/ns/net
lrwxrwxrwx 1 root root 0 Dec 29 00:03 /proc/21771/ns/net -> net:[4026531957]
vagrant@test:~$ sudo ls -la /proc/21775/ns/net
lrwxrwxrwx 1 root root 0 Dec 29 00:03 /proc/21775/ns/net -> net:[4026531957]
vagrant@test:~$ sudo ls -la /proc/21778/ns/net
lrwxrwxrwx 1 root root 0 Dec 29 00:03 /proc/21778/ns/net -> net:[4026531957]
vagrant@test:~$ sudo ls -la /proc/21764/ns/net
lrwxrwxrwx 1 root root 0 Dec 29 00:03 /proc/21764/ns/net -> net:[4026531957]
vagrant@test:~$
```

9.2.1 How Would Our Kernel-level Scenario Look Then?

The final "scenario" can be seen in figure 9.2.1.



So, to run `telegraf` only on the switches we would just launch it on the default network namespace! This can be done with a single `telegraf` process since the useful interfaces are all in the same Network namespace.

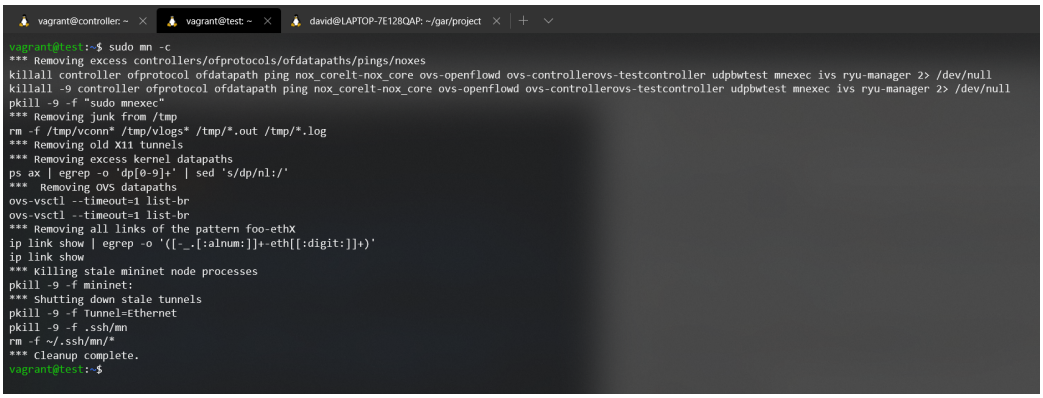
10 Troubleshooting

- If we are to use a terminal emulator without an **X server** installed or properly configured the **miniedit** tool will not run. This tool presents us with a GUI we can use to define our network and then it'll generate a script that brings it up for us. If we are to reroute the **stdout** of a VM we will need to set the `$DISPLAY` variable accordingly as **miniedit** used **tkinter** and it needs it to run correctly.

- If there are problems when launching the scenario try to clean up the previous environment. If we exit the mininet CLI by issuing the `quit` everything should be deleted correctly, otherwise we can always clean it up ourselves by running:

  `sudo nm -c`

  . You can take a look at the output in figure 10.



11 Appendix

We have decided to prepare an appendix so that we can shed some light on obscure topics not directly related to the project itself. We'll talk about tangential components of the project so that we can have a clearer idea of what's going on in the background and you can get a better grasp of the tools we have employed. It's a win win!

11.1 The Vagrantfile

I bet you have heard about `Virtualbox` this wonderful program lets us virtualize an entire computer inside our own so that we can try new Linux-based distros, use a Windows OS from Linux or just "create" a server farm for our own personal needs amongst many other use cases. These "virtual computers" are called **Virtual Machines** or **VM**s in `Virtualbox` lingo. The "bad" thing is that `Virtualbox` only offers a **GUI** (**G**raphical **U**ser **I**nterface) to manage new and existing VMs which makes the process extremely slow and changes it with each new update (the window titles vary, the menus are in different places...). This poses no problem at all to the average user but it becomes a nuisance in scenarios like ours.

Another point of concern is the VM's provisioning: How can we get files from the host machine into the VM? We commonly used shared folders between the host machine and the VM but the set-up process can be a real pain. Is there any hope left in the galaxy? Yes: Help me `Vagrant`, you are my only hope!

We can think of `Vagrant` as a wrapper for `Virtualbox` that let's us describe the VM's we want in a file called the `Vagrantfile`. We then run `Vagrant` with this file as an input and everything will be set up for us! By changing the `Vagrantfile` we can modify every VM in our topology. This includes provisioning new files, changing their memory, hostname, OS... This allows for a much more reproducible environment and hence a great portability.

The `Vagrantfile` itself is written in `ruby`. It's contents are mostly in plain English and we have included comments for the tricky parts so as to make everything as clear as possible. You can even use this `Vagrantfile` as a template for your own projects!

11.2 File Struct: `stdout` and friends

What's a file struct? We can think of it as an information bundle describing a place we can write data to and read data from. We can employ these file descriptors to communicate our programs with the exterior world by means of a file. In `C` we can open files through their file descriptors which we create thanks to the `fopen()` function. If you take a closer look at the documentation you will see the type returned by `fopen()` is in fact a pointer to a `FILE struct` (i.e a `FILE*`). This `FILE struct` contains info about the file itself: Have we reached the `End Of File` mark?, where are we going to read/write with our next instruction?, has there been any error when reading/writing data? This will let us handle our file in any way we want!

If you think about it we are constantly writing to the terminal from our programs using functions like `printf()` in `C` and `print()` in `python3`. Do you remember opening a file descriptor to be able to write to the terminal? I bet not! This is because our running programs are given 3 default file descriptors: `stdout`, `stdin` and `stderr`. These are connected to the terminal running the program (usually), the keyboard and the terminal as well (usually) respectively. If you have used `C` you may go ahead and try to call `fprintf()` and pass `stdout` as the file descriptor (the first argument). You'll see that you'll be writing to the screen! We can then see how both `stdout` and `stderr` are output file descriptors but `stdin` is used for reading keyboard input. As we are mainly concerned with `stdout` we won't go into much detail here.

Why do we have two file descriptors "attached" to the terminal you ask? This let's us separate a programs terminal output into 2 classes: normal output and error/debugging output. Even though both would appear in the terminal if we didn't take any further action we can redirect `stderr` to a file for later inspection which is a common practice. This redirection is carried

out when invoking the program from a terminal. The following command would redirect `My_prog.ex`'s `stderr` output to a file called `I_mesed_up.txt`:

```
./My_prog.ex 2>I_messed_up.txt
```

Now, a file struct is an abstraction used by **C**. We've previously said that `fopen()` returned a pointer, that is, a memory address. The things is that Linux itself knows nothing about a `FILE` struct, it only understands file descriptors which are a simple integer. Now the key aspect is to find out the relation between these `FILE` structs and the corresponding file descriptors. Even though we haven't dug any deeper we believe there must be some kind of "map" or "vector" somewhere that can be indexed with each file descriptor to get the address of each file struct. We may be outright wrong, but finding a `FILE**` somewhere whose indexes are file descriptors wouldn't be much of a surprise. Anyway, we have some predefined file descriptors we can use when invoking commands. These are:

- `stdin`: 0
- `stdout`: 1
- `stderr`: 2

You can even redirect a file descriptor to the place where another is pointing. Take care with the order used to carry out these redirections! The following would redirect `stdout` to where `stderr` is pointing:

```
./My_prog.ex 2>I_messed_up.txt 1>&2
```

We hope to have shed some light on how file descriptors work, what they are and how to use them!

12 Authors

- **David Carrascal**: GitHub Profile
- **Adrián Guerrero**: GitHub Profile
- **Artem Strilets**: GitHub Profile
- **Pablo Collado**: GitHub Profile

References

[1] Lars Buitinck, Gilles Louppe, Mathieu Blondel, Fabian Pedregosa, Andreas Mueller, Olivier Grisel, Vlad Niculae, Peter Prettenhofer, Alexandre Gramfort, Jaques Grobler, Robert Layton, Jake VanderPlas, Arnaud Joly, Brian Holt, and Gaël Varoquaux. API design for machine learning software: experiences from the scikit-learn project. In *ECML PKDD Workshop: Languages for Data Mining and Machine Learning*, pages 108–122, 2013.

[2] R. L. S. de Oliveira, C. M. Schweitzer, A. A. Shinoda, and Ligia Rodrigues Prete. Using mininet for emulation and prototyping software-defined networks. In *2014 IEEE Colombian Conference on Communications and Computing (COLCOM)*, pages 1–6, June 2014.

[3] InfluxDB. Manual influxdb.

[4] InfluxDB. Manual influxdb python api.

[5] Brian W. Kernighan and Rob Pike. *The UNIX Programming Environment*. Prentice Hall Professional Technical Reference, 1983.

[6] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice Hall Press, USA, 1989.

[7] Michael Kerrisk. *The Linux Programming Interface: A Linux and UNIX System Programming Handbook*. No Starch Press, USA, 1st edition, 2010.

[8] Kokila RT, S. Thamarai Selvi, and K. Govindarajan. Ddos detection and analysis in sdn-based environment using support vector machine classifier. In *2014 Sixth International Conference on Advanced Computing (ICoAC)*, pages 205–210, Dec 2014.

[9] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[10] Telegraf. Manual telegraf.