Final Year Project

# Service Function Chaining Strategies for Resource Constrained Edge Nodes

Chee Guan Tee

Student ID: 18202044

A thesis submitted in part fulfilment of the degree of

**BSc. (Hons.) in Computer Science**

**Supervisor:** Professor Madhusanka Liyanage


UCD School of Computer Science

University College Dublin

April 30, 2022

# Table of Contents

# Abstract

Edge security is becoming increasingly important in the modern era of edge computing. Therefore, to secure edge devices, new approaches must be developed to secure them to prevent malicious hackers from compromising them. The proposed solution to this problem is to deploy security services closer to clients and users, which decreases latency and can be deployed at a fairly low cost. By utilising the service function chaining (SFC) strategy, multiple security services such as Snort and Suricata can be chained together to provide better detection and protection against network attacks. In this paper, a SFC strategy is implemented on a Raspberry Pi VM using the Docker virtualisation technology and Open vSwitch to serve as a security device that can be easily deployed and managed on the edge network. From experiments, it is shown that this architecture provides adequate performance for smaller-scale networks with lower network speeds. With the accumulating packet speeds however, the performance of the system degrades and attacks will less likely be detected and prevented by the security services.

# Chapter 1: **Introduction**

The Internet of Things (IoT) has recently been a hot topic since it is being widely used for a variety of applications and devices, including wireless sensors, medical equipment, sensitive home sensors, and security services. Service Function Chaining (SFC) is widely used in large-scale networks with sufficient computing power, but there has been little research into applying SFC in resource-constrained environments. In addition, a single security service is insufficient, and hence we can utilise SFC to chain multiple services together. For this project, the idea is to launch multiple security services (Snort, Suricata, etc.) on a Raspberry Pi. It facilitates SFC by chaining services together to select and steer data traffic depending on the packets we receive, or in other words, handles traffic classification. Ideally, the services will be deployed in Docker containers and orchestrated using a container orchestration tool like Docker Swarm or Kubernetes (K3s).

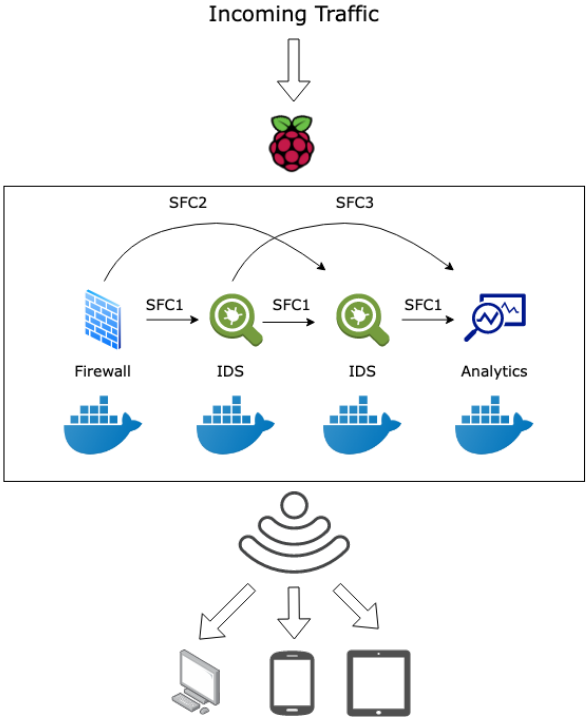



Figure 1.1: Basic Architecture of a SFC Security Strategy

However, the main focus of this project is to provide a working SFC implementation of the security services, with additional services being added as time passes. The main challenges for this project include running multiple services within a resource-constrained device and implementing service function chaining correctly as illustrated in Fig. 1.1. Overall, multiple approaches have been proposed and it has been decided that Open vSwitch and Docker are both more suited for this project compared to other alternatives. The final implementation is deployed on a virtual machine with similar specifications to a Raspberry Pi, and packets are able to be routed dynamically through the architecture, which meets our goal for this project. Much thought has been put into the design and testing of the final implementation to ensure that packets are transmitted correctly along the designated path.

# Chapter 2: Project Specification

## 2.1 Datasets

To test our implementation, we used some sample PCAP files with pre-generated traffic. The PCAP files are used to simulate malicious packets for testing purposes. Ideally, different PCAP files with malicious UDP/TCP traffic will be used to evaluate our system. Our goal is to create good SFC chains that have high detection rates, so a variety of different PCAP files should be used for testing to get a better picture of the overall performance of our system. Some example traffic categories include:

- Malware Traffic
- Packet Injection Attacks
- PCAP files from capture-the-flag (CTF) competitions and challenges.
- SCADA/ICS Network Captures

and can be obtained from [1].

## 2.2 Resources Required

### 2.2.1 Software

- Gitlab Code Repository, https://csgitlab.ucd.ie/jasontcg/FYP
- VirtualBox 6.1.34, for emulating Raspberry Pi
- kubuntu 22.04, underlying image for VM
- Docker 20.10.14, for containerizing security services
- Portainer 2.11.1 for managing Docker containers
- Python 3.9.7, for executing routing script and firewall
- scapy 2.4.5, Python 3 library for routing packets
- tcpreplay 4.3.2, for replaying traffic in client container
- Snort 2.9.19, one of the IDS
- Surata 6.0.5, one of the IDS
- Open vSwitch v2.16.0, for implementing SFC

### 2.2.2 Hardware

- Raspberry Pi, unavailable due to supply chain issues
- Personal laptop, Macbook Pro 2019 (1.4GHz Core i5, 16 GB RAM)

# Chapter 3: Related Work and Ideas

## 3.1 Core Concepts

### 3.1.1 Service Function Chaining

SFC is a concept developed to allow for the provision of dynamic and flexible services in softwarised networks. Essentially, service function chains are a sequence of multiple virtual network functions (VNFs) that enables traffic steering [2]. A typical SFC architecture includes several components, such as VNFs, service function chains, service classifiers, and an SDN controller, as shown in Fig. 3.1.

A VNF operates on network packets on multiple layers of the networking stack. In the context of this project, our VNFs will be security services such as Snort and Suricata [3]. A service function chain is a sequential chain of one or more VNFs that helps to automate traffic within a network. With service function chains, we can define different sequences of services and handle packets according to their headers and metadata. By using the flow classifier in open source libraries such as OpenDaylight (ODL), we can correctly classify network traffic based on a series of policies. Finally, the SDN controller acts as the master node of the network and determines where a packet will go next. To elaborate, VNFs communicate with the controller to determine the next destination of a packet.

By using SFC, multiple VNFs can be linked together sequentially [4], and this allows the chaining of multiple security services together to provide a higher security level for our network. For this project, an SFC classifier can be used to classify the different packets, and by using an SDN controller, we can coordinate traffic to and from other services.

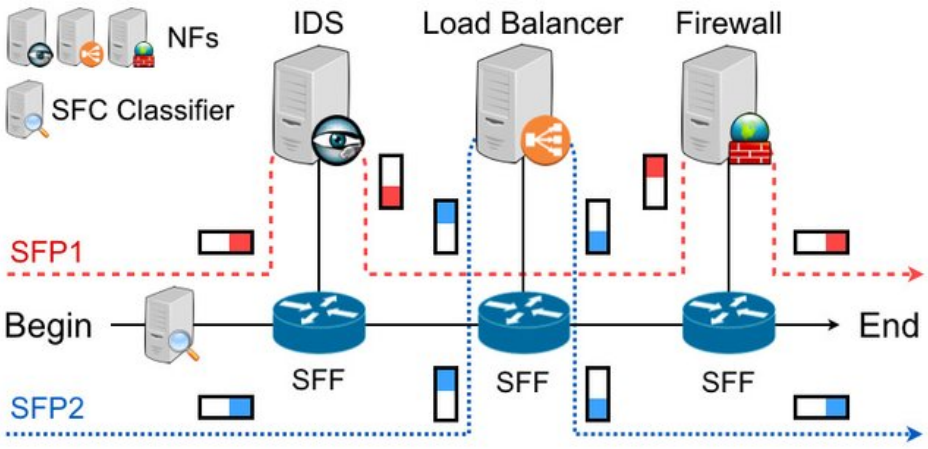



Figure 3.1: SFC Architecture [5]

Traditionally, we would need to define packet routing in each node, which adds complexity because it is difficult to maintain and change routing behaviour so that the entire system functions as expected. In other words, if we need to change the routing behaviour of individual nodes, we must do so in the individual nodes, whereas in SFC, we can easily change it through configuration files in the SDN Controller. SFC is suitable for this project as it allows us to define the routing of

packets easily depending on their packet header.

### 3.1.2 Resource Constrained Edge Nodes

Resource-constrained nodes are lightweight devices with limited computing power and storage capabilities. They are widely used in Internet of Things (IoT) applications due to their affordable cost and the redundancy that they create.

Edge computing is essentially a computing model that delegates the computing and processing of data to edge nodes that are close to the users instead of doing all computations in the cloud [6], which is illustrated in Fig. 3.2. Since IoT devices (surveillance cameras, mobile devices, etc.) generate a lot of data, we will be overburdening the cloud servers to process all this data. Besides, as a cloud is usually located on a remote server, latency is imminent as the devices need to constantly communicate with the cloud for every operation. Significant bandwidth and processing can be reduced with low latency by delegating a portion of data processing to the users on edge nodes. For instance, edge computing plays a key role in 5G networks, and according to IDC's data times 2025 report, "50% of data will be analyzed, processed, and stored on the edge of the network by 2025."[6].

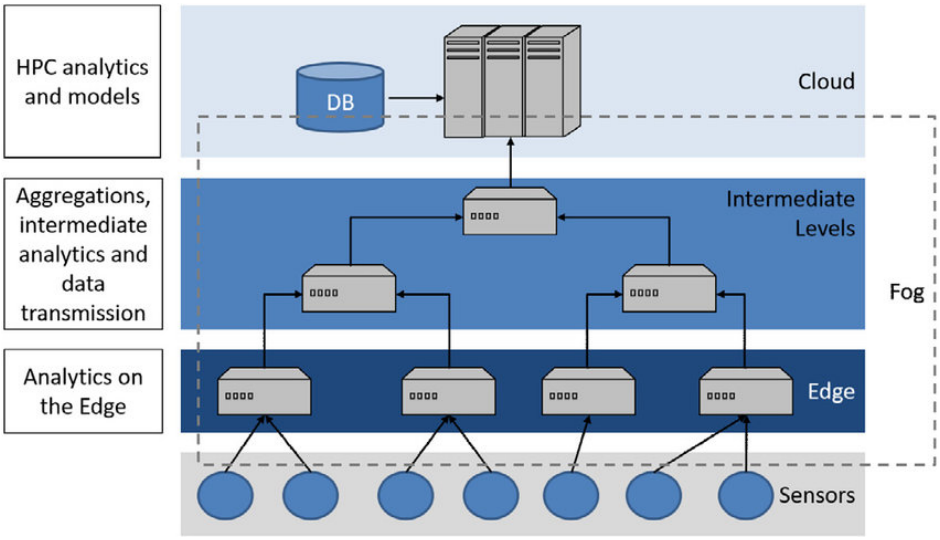



Figure 3.2: Edge Computing Architecture [7]

In this project, we are particularly concerned about implementing a security edge node that provides a variety of security services at the edge layer. This is beneficial in terms of edge computing, as edge nodes are located closer to users; we can collect data packets with minimal latency from many IoT devices and hence make better security decisions from those data packets [8]. As IoT devices are usually resource-constrained and do not have enough resources to deploy and manage a firewall on their own, we can implement an edge-based firewall such that it filters out unwanted traffic to and from the device. To elaborate, we can have a set of firewall rules on the edge node which determines which packets are allowed to pass through [8]. Intrusion detection is also important, as, if IoT devices get compromised, we need to detect it so we can limit the damage caused by the attack.

There are a lot of choices for our edge nodes to choose from, such as Arduino, Raspberry Pi, Phidgets, etc. The Raspberry Pi is an ideal choice for the resource-constrained edge node as, despite its small size, it's a powerful and cheap device which fits into our use case [9]. In this paper, we'll look at the **Raspberry Pi 4**, which is the latest version of the Raspberry Pi product. In addition, the Raspberry Pi supports 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless and Gigabit

Ethernet, which fits our purpose since, in later stages, we might need to test it with other devices on the same network. Besides, it can also have a large RAM (2GB–8GB) which can be utilized to run Open vSwitch and other security services concurrently. In addition, it runs on Linux natively, which is perfect for security services such as Snort and Suricata. Lastly, we can also easily host web servers on the Raspberry Pi, which helps in providing a web-based monitoring platform for our edge node.

### 3.1.3 Docker

Docker is a virtualization tool that allows us to run different applications in containers that are isolated from each other [10]. Docker containers only contain application-specific dependencies and libraries, which makes them more lightweight than virtual machines, as they run on separate guest operating systems. Docker is preferred to virtual machines in a lot of use cases as virtual machines often have higher I/O latency, which affects performance and throughput [10]. Besides, Docker containers can be easily created through Docker images, and it's also common to have one single application within a Docker container, which can reduce dependency clashes. In addition, Docker images can be assembled through Dockerfiles, and we can spin up multiple instances from a single Docker image. Process isolation that Docker containers provide is useful as each application is isolated from the other, so in the event of a security breach of one of the containers, the others won't be affected.

In the context of this project, Docker will be useful as it allows us to containerize different security and monitoring services. Besides, having the whole edge node as a Docker containerized image will also be useful since the re-deployment or replication to a new Raspberry Pi can be achieved with minimal effort. In terms of networking, we can utilize Docker Swarm to manage container networking with ease. Running MicroK3S is also a potential solution to managing multiple containers and handling scaling issues.

### 3.1.4 Intrusion Detection Systems

Traditionally, firewalls are used to block malicious traffic by analysing the packet header data such as the source, destination, and port number and checking them against a security policy. However, a malicious payload can contain the correct headers and pass through the firewall without issues. Therefore, IDPS systems are needed to examine the actual payload of packets to detect and prevent potentially malicious packets. There are two categories of IDS, mainly signature-based intrusion detection systems (SIDS) and anomaly-based intrusion detection systems (AIDSs) [11].

Signature-based intrusion detection systems use pattern matching techniques to match logs against known attacks. In other words, it checks current logs to see if they match the signature database. Open-source tools such as Snort and Suricata use a signature approach to detect intrusions. Although this approach performs quite well in practice, it performs poorly against zero-day attacks, which are attacks that have not been seen before. On the other hand, anomaly-based intrusion detection systems attempt to solve this by learning the normal behaviour of the network and detecting anomalies or abnormal behaviour. By using normal traffic as the training data, AIDS can learn to identify patterns such as zero-day attacks that differ from normal traffic. Besides, internal malicious behaviours can also be discovered through AIDS [11]. However, a high false-positive rate might be a concern, and there are no common open-source tools for AIDS, so a custom model needs to be created to support this approach.

In this project, Snort will be used as one of our security services since it is a widely-used open-source tool that supports IDS and also acts as an intrusion prevention system (IPS). Users can

define a set of rules/policies known as Snort rules that allow logging and, based on the specified condition, [12]. Besides, real-time traffic analysis and logging can be performed easily on IP-based networks. Suricata is another open-source tool similar to Snort but supports multi-threading and can process traffic much faster than Snort in a multi-threaded environment, which is ideal for a Raspberry Pi due to its multi-core processor.

## 3.2 Literature Review

This section focuses on the research I have conducted in relation to service function chaining on resource-constrained devices.

### 3.2.1 Fog Computing based security service function chaining

Liyanage et al. in [4] emphasized that a single security service is insufficient in providing ample protection, so SFC can be utilized to combine multiple security services. This paper introduces the idea of an Edge Level Security Orchestrator (SO) which handles logging and resource allocations for virtual network functions (VNFs) and also supports multi-tenant access capabilities. Besides, the security fog node with SFC is implemented using Open vSwitch and assigns a unique IP address to each security service container. A firewall is also used to provide basic firewall protection in the proposed architecture in Fig. 3.3. The author containerizes individual security services in Docker containers, and the SO handles packet routing and implements SFC. The architecture is deployed on a Raspberry Pi 4 and a Raspberry Pi VM for evaluation purposes to show that it can be deployed to different platforms. Notably, the author found that by deploying multiple instances of the same security services (Snort and Suricata) and increasing the data speed (packets per second), CPU usage increased significantly and packet drop rates started to increase. It is also notable that Snort seems to perform better than Suricata, with fewer packet drops when data speed is increased. Overall, it is clear that the SFC architecture can handle high packet rates and performs quite well in practice.

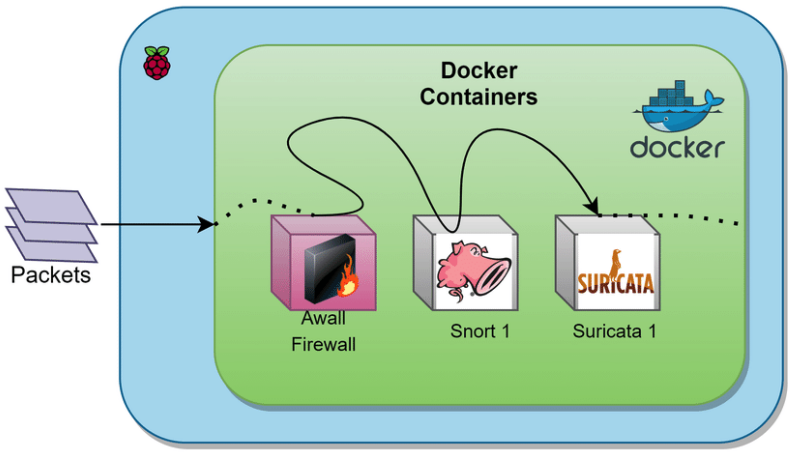



Figure 3.3: Proposed Architecture for Fog node [4]

The Fog layer lies above the edge layer and this research provides a proof of concept that SFC can be done on a Fog node with good performance. As the research was also conducted on a Raspberry Pi 4, it would be a good baseline to compare our research to evaluate the outcome.

On the other hand, the author did not go into the scaling techniques of the individual security services, and the deployment method was not discussed in detail. In this paper, scaling methods, deployment and the SFC routing approach will be looked into in detail and investigated.

### 3.2.2 Container-Oriented Edge Management of Virtual Security Functions

Boudi et al. in [13] argued that deploying security applications within lightweight Docker containers is preferable compared to deploying them on virtual machines or bare-metal systems. In this paper, the author mentioned that Docker is a fast and efficient tool to deploy containers and has less overhead compared to virtual machines as they can all run on the same kernel of the host machine. Besides, it is emphasized that Docker containers can be easily scaled and can be deployed in a separate environment. High manageability is also a benefit, as the manager node in a Docker swarm architecture can provision and manage security services on demand. The authors found that systems on bare metal (SoBM) processed a similar number of packets compared to systems in a Docker container (SoDC). In terms of network utilization, SoBM performs slightly better than SoDC, but not by a large margin. Interestingly, the number of packets dropped is lower on SoDC, potentially due to the lower CPU utilization rate on SoDC and the higher number of packets processed in SoBM.

However, it is worth noting that the experiment was performed on a single containerized instance of Suricata, and performance will likely be affected when multiple containers need to communicate through a bridged network. This research paper demonstrates that running security services in Docker containers generally does not impact performance and brings the variety of benefits discussed above. So, in this project, Docker will be used to containerize the security services used in our edge node.

### 3.2.3 Intrusion detection systems on the Raspberry Pi

In [14], Sforzin et al. carried out research which investigated the feasibility of running Snort on a Raspberry Pi 2 Model B. The paper mentioned that in IoT environments, security plays a vital role, and the idea is to have a small device that can be easily deployed with minimum configuration. The authors also proposed a scalable architecture with multiple Rasberry Pis that can share and exchange data between each other either through Zigbee[15] or low-power Wifi. The authors found that small packet sizes have higher CPU usage and packet drop rates since interrupts happen more frequently and start to affect CPU performance. Besides, by having a smaller rule set, performance is increased, albeit with less security. It is also worth noting that by increasing the number of rules, RAM usage also increases since Snort needs to load those rules into memory. Besides, small packets drop much quicker when the data rates increase compared to long packets. In addition, the number of alerts reduces when the data rate gets higher because more packets are getting discarded, and fewer packets are getting inspected.

This paper gives us a rough estimate of the performance when running security services on the Raspberry Pi and will be useful during the final evaluation of our research. However, the model used in this paper is the Raspberry Pi 2 Model B, which has lower CPU, RAM, and connectivity speeds than the latest Raspberry Pi 4. In addition, the latest model supports Wifi, which is absent from the Raspberry Pi 2 Model B. Hence, we can expect better performance and lower packet drop rates when running security services on the Raspberry Pi 4.

In [16], Imrith et al. also conducted a similar experiment but with multiple instances of container-

ized Snort and Suricata on the Raspberry Pi 4. In this paper, it is shown that performance is indeed better on the Raspberry Pi 4, and CPU and RAM usage are lower due to more powerful hardware. It is also clear that when multiple Suricata instances are run concurrently, the CPU and RAM usages increase, but the performance across the instances remains constant, indicating that scaling is possible. However, when running multiple instances of Snort, the alert percentage decreases when the number of instances increases, which suggests that CPU allocation plays an important role.

### 3.2.4 Dynamic orchestration of security services

In [16], Imrith et al. proposed an architecture that supports the dynamic orchestration of security services. In Fig. 3.4, the security orchestrator (SO) is responsible for configuring the type of security services to deploy and the lifecycle (start, stop, terminate) based on demand. Besides, Open vSwitch (OVS) is used to assign individual IP addresses to different services. The SO is responsible for handling multiple device support such that the packets from different devices can be differentiated and handled by the security services. Additionally, the SO should also perform load balancing and performance optimisation such that packets are routed efficiently to the best service. The paper also mentioned the traffic classification feature of the SO, which classifies the traffic by TCP/UDP port numbers to get a better quality of service (QoS). The authors also mentioned the dynamic allocation of resources of service instances as one of the responsibilities of the SO, which functions by allocating a certain amount of memory and CPU power to individual containers and can be changed when load increases.

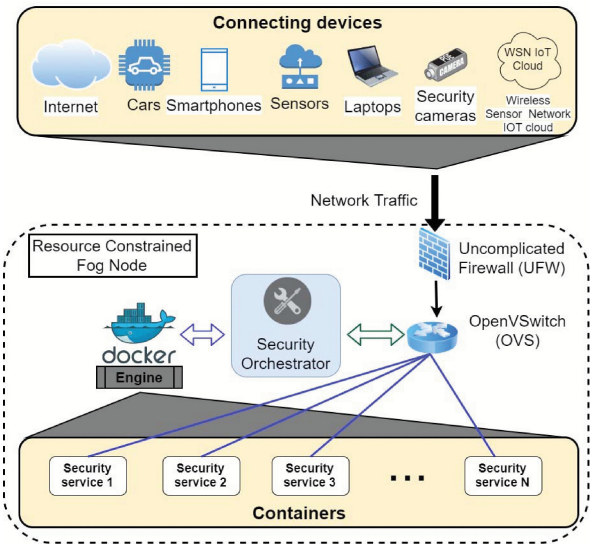



Figure 3.4: Proposed Architecture for a Fog node [4]

Although this paper does not present an actual implementation of the SO and the configuration of OVS, it gives us a good idea of the whole infrastructure and the expected behaviours of each component.

### 3.2.5 Open vSwitch with containerized VNFs

In [17] Bonafiglia et al. evaluated the performance of deploying VNF chains on a single server using a vSwitch. It mentioned that Docker containers have different network namespaces and can only communicate with the vSwitch that lives in the host network namespace using $veth$ pairs.

The paper showed that when the number of containers increases in the chain, the throughput decreases quickly, which might be an issue if we have long VNF chains. Besides, the authors concluded that Docker containers show acceptable results when the VNFs are tied to a specific process and provide better latency than VM-based approaches.

Although this research was done on a server, it showed us that running VNFs in containers is a viable approach and provides better latency than VMs.

# Chapter 4: Design Considerations

## 4.1 Current Architecture

The whole architecture of this project is built on a Lubuntu virtual machine running on VirtualBox, and several aspects of the architecture will be detailed in-depth in the sections below: SFC, Client, Firewall, Snort, and Suricata. All of the source code for this project can be found on my personal Gitlab repository at [18]. In Fig. 4.1, we can see that four Docker containers are linked together using Open vSwitch bridges to allow manual packet routing decisions within each container. Traffic is generated from the client container, which is then routed via the SFC chain until it reaches the Suricata container at the end. It should be noted that ICMP packets (typically generated by tools like `ping` and `traceroute`) are routed directly from the firewall to Suricata via a separate chain, and this behaviour can be changed to accommodate other rule-based routing requirements. Originally, a container with ntopng was included at the end of the chain but was ultimately removed due to excessive resource consumption. Besides, the project's final implementation closely resembles the planned design in the introduction section and is sufficient to demonstrate the capabilities of SFC in a resource-constrained environment.

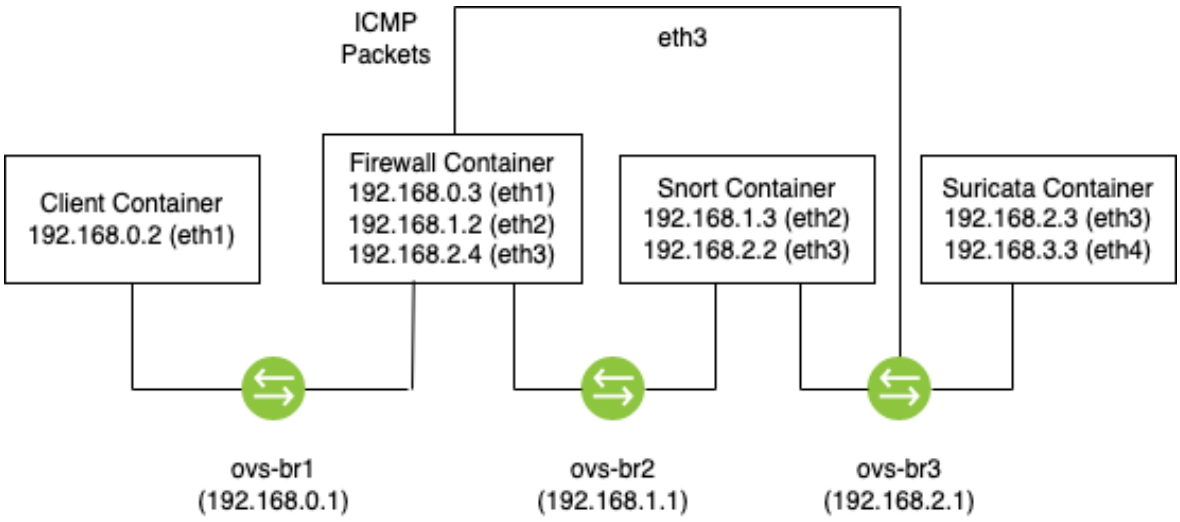



Figure 4.1: Overview Architecture

Although the project is implemented on a virtual machine, it can be deployed to a Linux-based Raspberry Pi system with relative ease since the VM is configured with similar specifications to a Raspberry Pi. This SFC chain implementation is based on security services but can be extended to other usages as well, such as HTTP video optimization [19] and QoS provisioning [20].

## 4.2 Open vSwitch

The virtual networking layer may use Open vSwitch to act as a virtual switch in VM settings and offers standard control and visibility of the interfaces. Although the fast and reliable built-in

Linux bridge can be used to bridge containers together, it is harder to maintain and, because of the dynamic nature of our application, which requires rapid Docker container manipulations, Open vSwitch is preferred in our architecture [21]. Open vSwitch allows us to create virtual bridges which can be used to connect Docker containers. Containers connect to virtual ports on the virtual bridges, and each container will also have a virtual network interface. For example, in Fig. 4.2, we can see that both containers are connected using an OVS bridge and they communicate with each other using their respective network interfaces (vnic-1 and vnic-2). To elaborate, within each container, we can sniff traffic from one interface and replay or forward it to another interface. This essentially forms a chain through which packets travel. Besides, an important note is that we do not detach the containers from the docker0 bridge as we still require internet access within each container, and the default docker0 Linux bridge allows us to retain network connectivity with the outside world. Similarly, the same paradigm can be applied to multiple virtual network interfaces and chains, which is shown in Fig. 4.2.

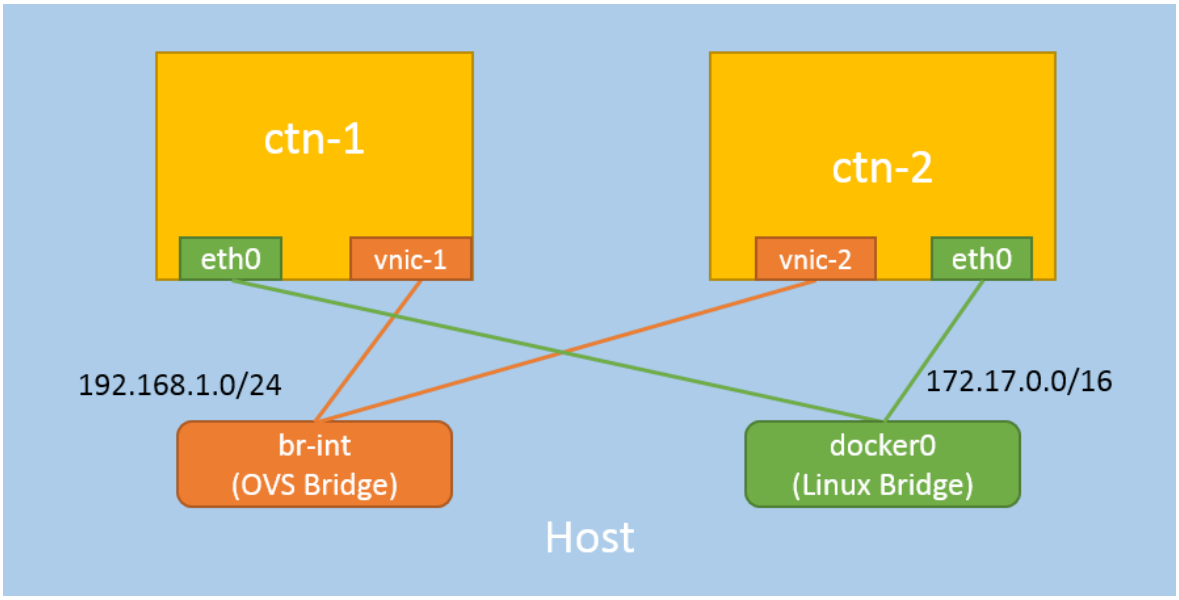



Figure 4.2: OVS Bridge between two containers [22]

## 4.3 Explored Alternative Strategies and Dismissing Reasoning

There are a few alternative strategies that has been explored before settling on the current architecture.

### 4.3.1 Kubernetes

Initially, Kubernetes is considered instead of using Open vSwitch directly, but this proves to be a non-trivial task. Even though Kubernetes Container Network Interfaces (CNIs) enable network connection inside Kubernetes clusters and, in certain situations, may also provide security and management by setting particular network regulations, they lack the essential characteristics to allow Virtual Network Functions integration [23]. To elaborate, since Kubernetes does not provide

native support for having two or more network interfaces in a container and SFC chains are not supported natively, we can only opt to use a Network Service Mesh architecture (see [24]) within Kubernetes. NSM operates by enabling pods to communicate with one another by providing linkages between them. For example, in Fig. 4.3, an SFC chain is constructed using the NSM architecture, which allows them to connect on the network/data link layer. This fits our use case

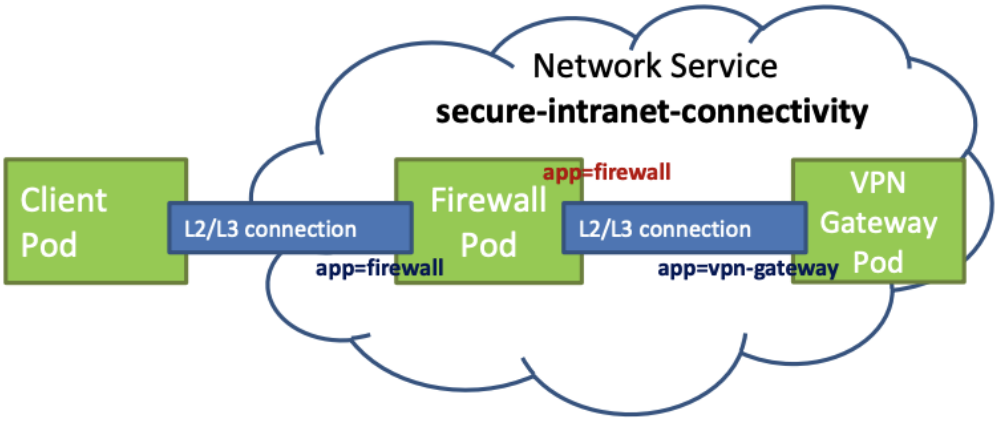



Figure 4.3: NSM SFC Chain[23]

as L2/L3 connections are required to relay packets between our security containers. However, if we look at the components of NSM in Fig. 4.4, it is clear that the architecture is not an ideal implementation for our use case, largely due to the additional performance impact of running all of the components within a resource-constrained node. In addition, compared to the Open vSwitch solution, a large amount of boilerplate code and configurations are required.

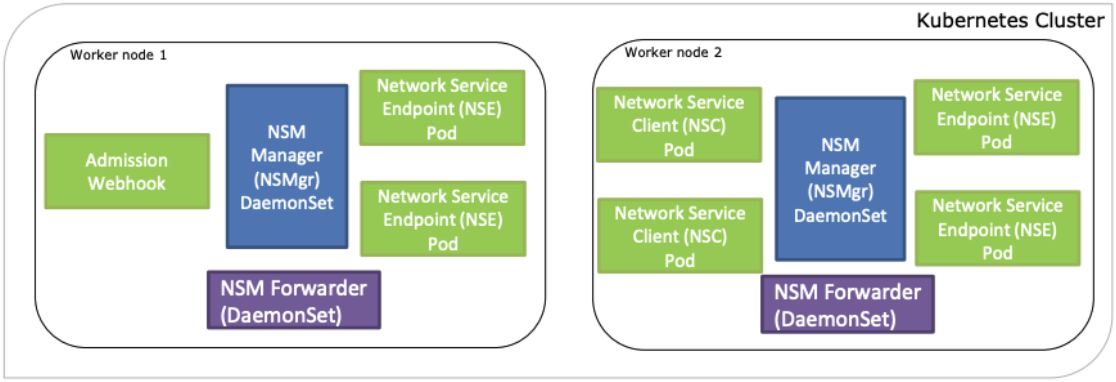



Figure 4.4: NSM Components[23]

In conclusion, this technique is better suited to systems with large-scale deployments across numerous Kubernetes nodes. Therefore, I have chosen not to continue with this implementation.

### 4.3.2 Manual Implementation of Network Bridge

Another approach I considered is to implement the network bridge myself. A fast Layer 2 Linux bridge can be created manually which allows us to communicate between containers [25], see Fig. 4.5. However, this proves to be quite cumbersome since multiple commands needs to be issued to create the bridge between containers. Furthermore, `ovs-docker` and `ovs-vsctl` already provide the necessary functionality to add bridges so it does not make much sense to go with this approach.

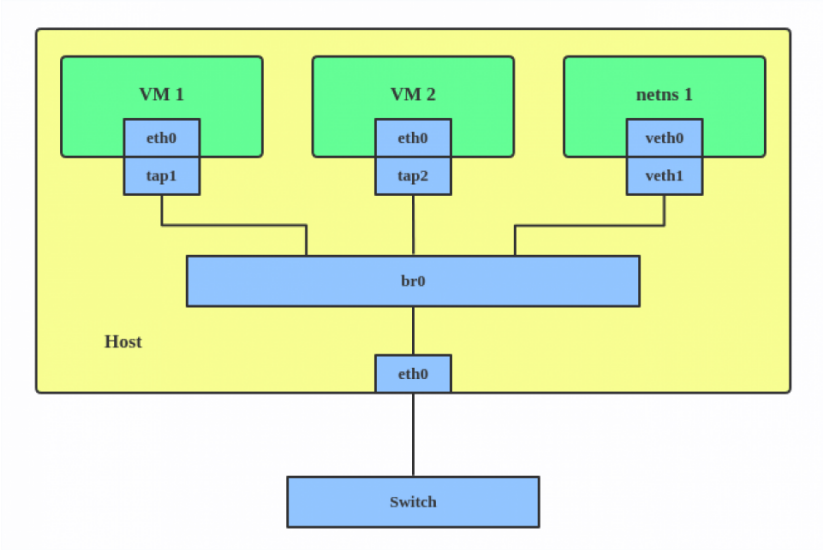



Figure 4.5: Linux Bridge[26]

# Chapter 5: Outline of the Proposed Approach

## 5.1 Service Function Chaining Implementation

Docker containers and Open vSwitch bridges created from the host container are used to implement the SFC architecture, and both components work together to construct service chains that enable network traffic to flow through them.

### 5.1.1 Docker Containers

Instead of using more minimal images such as `alpine`, I chose to use the `ubuntu` image for all of the Docker containers because it comes with a decent package manager out of the box and some utilities preinstalled, such as `wget`. The containers are started with the following command, and the `-cap-add=NET_ADMIN` flag is added so that we can later enable promiscuous mode on the containers' virtual network interfaces.

```
> docker run --name client --cap-add=NET_ADMIN ubuntu
```

After launching the container, we can then ssh into the container using the following command:

```
> docker exec -it client /bin/bash
root@eac31889098c:/
```

Besides, some common tools are also installed using `apt` for all of the containers in the service function chains.

```
> apt install net-tools inetutils-ping vim pip tmux iproute2 git
    tcpreplay tcpdump -y
```

After installing the essential tools, the FYP repository is cloned to each container using the following command:

```
cd ~
> git clone https://github.com/AmplifiedHuman/FYP.git
```

The reasoning for doing this is that I can apply changes directly to the files in the repository on my personal machine and sync the update changes in each container using `git pull`. To manage all the containers in a simple and intuitive way, I used the Portainer tool, which shows all the status of my containers using a web interface. Installation instructions can be found on the official documentation site in [27]. For instance, in Fig. 5.1, we can see all our containers are running and we can also manage the state of our containers easily. Alternatively, Docker commands can be issued to manage the state as well, but it usually requires multiple commands to do so, so I chose to use the Portainer dashboard.

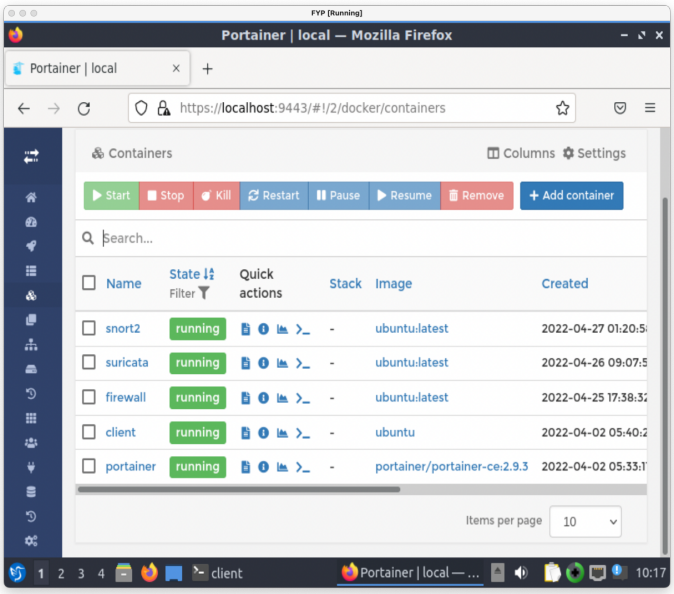



Figure 5.1: Portainer Dashboard

### 5.1.2 Open vSwitch Bridges

We can start by creating a bridge using the `ovs-vsctl` tool which updates the `ovs-vswitchd` configuration. Note that we also configure the IP address and net mask using `ifconfig`.

```
> ovs-vsctl add-br ovs-br1
> ifconfig ovs-br1 192.168.0.1 netmask 255.255.255.0 up
```

Next, we use a tool called `ovs-docker` which helps to connect each docker container to the new bridge through a port. This steps also creates a virtual network interface in each container to communicate with each other.

```
> ovs-docker add-port ovs-br1 eth1 container1 --ipaddress
   =192.168.0.2/24
> ovs-docker add-port ovs-br1 eth1 container2 --ipaddress
   =192.168.0.3/24
```

We can then verify that both containers are connected via the bridge by using the `ping` command on both containers. For example in container1:

```
> ping 192.168.0.3 -c 1
PING 192.168.0.3 (192.168.0.3) 56(84) bytes of data.
64 bytes from 192.168.0.3: icmp_seq=1 ttl=64 time=1.15ms
```

Next, we need to create a new flow for the new bridge so that all traffic can pass through the bridge. Note that this step is crucial and that without setting up the flow, we wouldn't be able to transmit TCP packets through the OVS bridge.

```
> sudo ovsc-ofctl add-flow ovs-br1 actions=ALL
```

To automate some of the manual commands, a `setup.sh` script is created to automate the addition or removal of ports on a bridge if there are any changes to the containers.

### 5.1.3 Interfaces

Interfaces in the containers must also be configured appropriately in order to record or analyze network traffic. First, interfaces should be put into promiscuous mode, which allows every data packet to be read by the virtual network interface, although it's not intended for the interface. This is especially important since we are replaying packets that are not in our local network.

```
> ip link set eth1 promisc on
```

Besides, network interfaces usually use techniques such as tcp-segmentation-offload (tso), generic-segmentation-offload (gso), generic-receive-offload (gro), and large-receive-offload (LRO) to speed up packet handling. This behaviour involves combining smaller packets into larger "super packets" and it must be deactivated since it breaks the dsize keyword and TCP state tracking for IDS/IPS systems like Snort and Suricata [28]. We can use the `ethtool` to disable them in one command as follows:

```
> ethtool -K eth1 gro off lro off tso off gso off
```

As both settings do not persist after container restarts, several scripts like `firewall_setup.sh`, `client_setup.sh`, etc. have been created to automate this process. We can list all interfaces and check if they are in promiscuous mode using `ifconfig` as shown in Fig. 5.2.

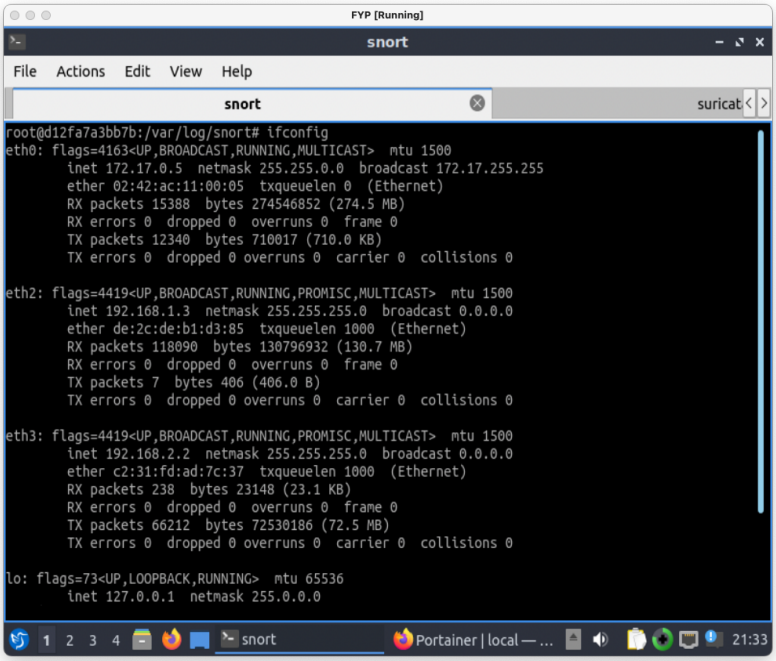



Figure 5.2: Show Interfaces

### 5.1.4 Packet Forwarding

Once the bridge has been configured correctly, we need to handle packet forwarding within the containers. The `scapy` Python library is especially often used for this purpose, as we can manipulate, sniff, and forward network packets programmatically. A background Python 3 script, `route.py` [18] is created to handle packet forwarding for intermediate containers in the service function chain.

```
def send_packets(packet):
```

```
    sendp(packet, destination_interface)

def start():
    sniff(iface=source_interface, prn=send_packets)
```

Listing 5.1: route.py

From the code snippet above, we can see that when the start function is called, the sniff function from `scapy` is executed and we start sniffing from the source interface. Once we receive a packet, the callback function send_packets is executed and the received packet is sent to our destination interface. For instance, in Fig. 5.3 packets first arrive through `eth1` and get captured by the script, the script then sends the packets to the output interface `eth2`. Note that we don't need to execute this script for the first and last containers (client and suricata).

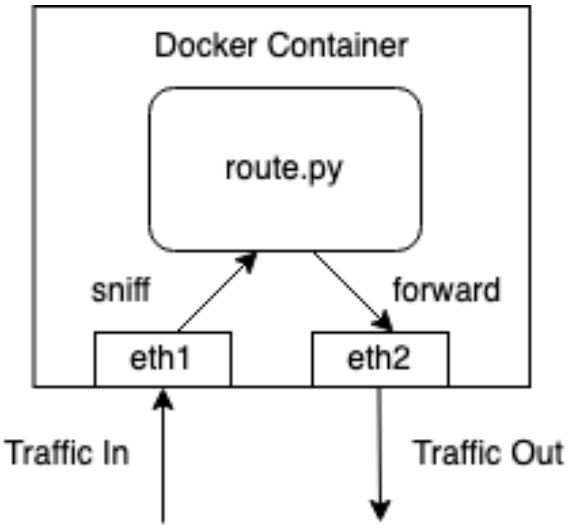



Figure 5.3: Packet Forwarding within Containers

## 5.2 Client

The client container is the first container in our SFC implementation and serves as the source of our traffic. It is connected through `ovs-br1` so we can replay traffic through `tcpreplay`, see Fig. 5.4. The `-i` option is used to specify the interface and `-K` preloads packets in RAM to increase packet replay speed while imposing a startup performance impact. Besides, we can also specify the `-M` option to cap the packet transfer speed at a specific speed (in Mbps). Alternatively, the `client.py` script can also be used to replay traffic from a given `.pcap` file.

## 5.3 Firewall

Packet-filtering firewalls operate at the OSI model's network layer, and they make processing choices depending on network addresses, ports, and protocols [29]. For our specific implementation, our firewall does not block packets but instead forwards ICMP packets directly to the Suricata container without passing through the Snort container. This is highly configurable and we could

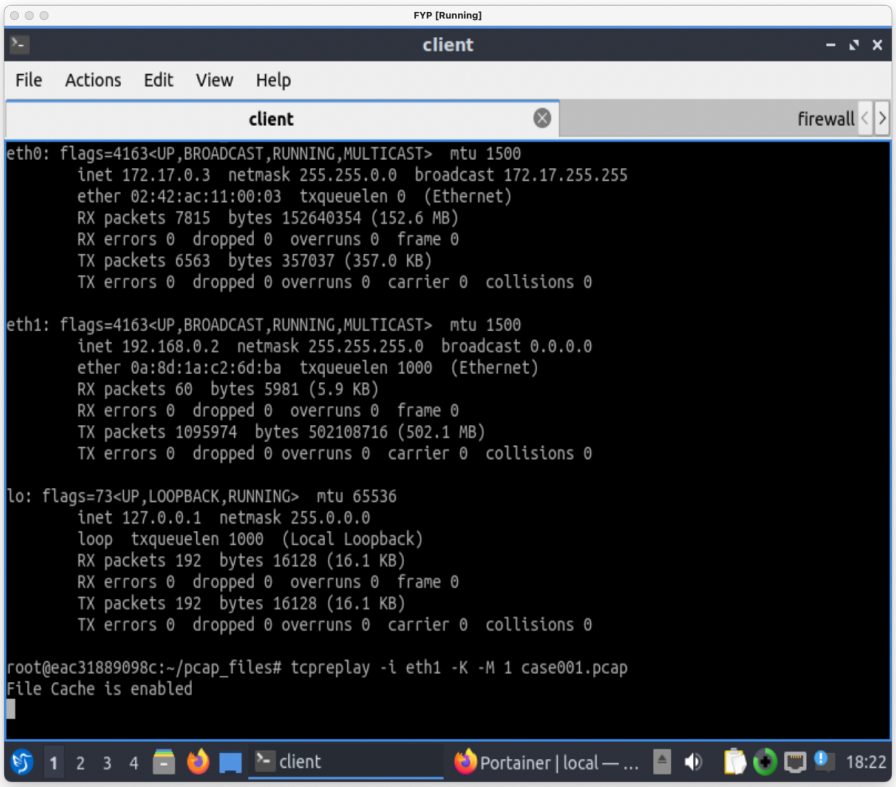



Figure 5.4: Demonstrating TCP Replay

easily change it to fit our needs. For example, with a little extra code, we can block traffic originating from a given IP or route different packet types (UDP, TCP, etc) through a different service function chain. The firewall container has access to 3 interfaces: `eth1` connects it with our client, `eth2` communicates with snort, and `eth3` connects it to the Suricata container. A Python 3 script, `firewall.py` is used to route traffic among the interfaces, and changes can be made to this script to accommodate other firewall rule changes.

```
def send_packets(packet):
    if ICMP in packet:
        sendp(packet, icmp_destination)
    else:
        sendp(packet, normal_destination)


def start():
    sniff(iface=source_interface, prn=send_packets)
```

Listing 5.2: firewall.py

Note that instead of having one destination interface, we have two destinations. We first check if the packet is an ICMP packet and if so, we forward it directly to the Suricata container otherwise we forward it to Snort. We can verify that this is working by executing the ping command on the client container (`192.168.0.3`) and ensuring that there are no ICMP packets (only ARP) reaching the Snort container (see Fig.5.6a) but we should be able to see packets on the `eth3` interface in the Suricata container (see Fig. 5.6b).

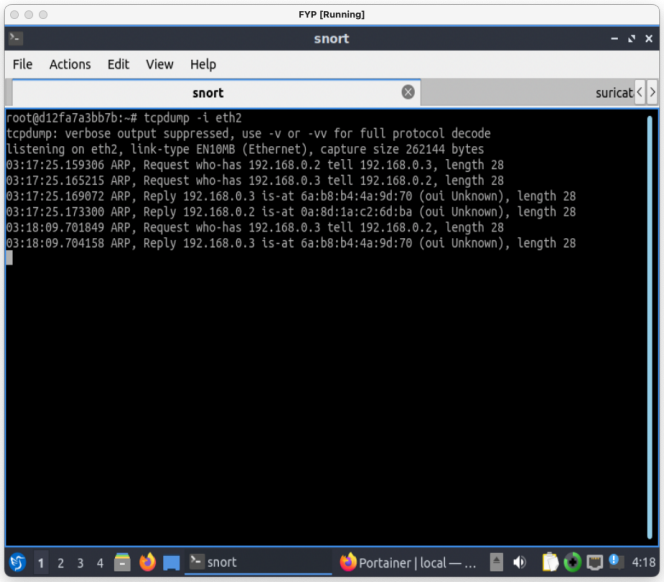



Figure 5.5: Snort Container eth2 traffic

## 5.4 Snort

We install Snort 2 from the source by following the Snort 2 guide and also installing the registered rules set. Next, we need to modify the following line in the `/etc/snort/snort.conf` file to accept traffic other than our home network.

```
ipvar HOME_NET any
```

The routing script `route.py` executes in the background which forwards packets from `eth2` to `eth3`. However, we cannot rely on `systemctl` to start services in the background since it is unsupported in Docker containers, [30]. To get around this, we utilize `tmux` to establish separate sessions for the routing script and Snort. To elaborate, we can run the routing script in the background using the following command:

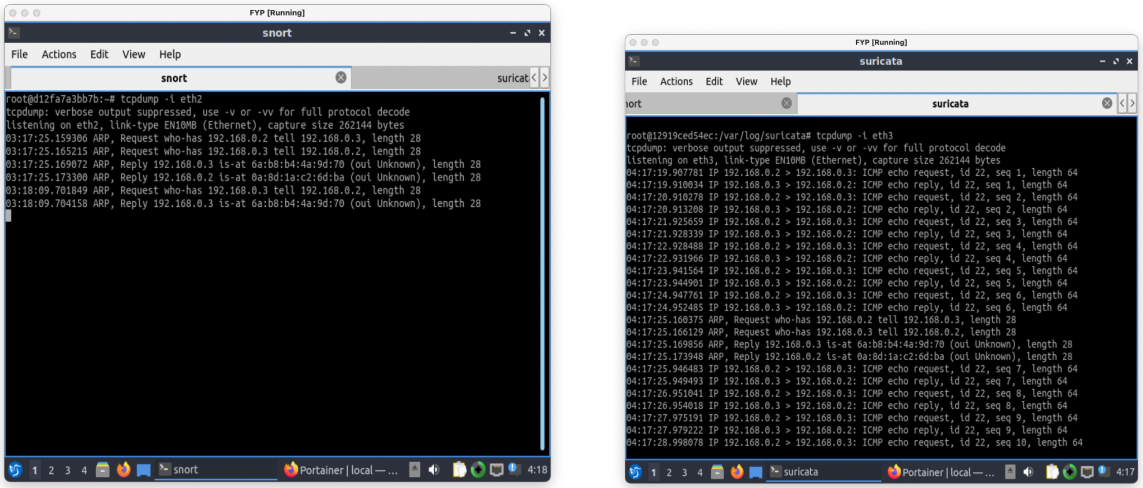
```
> tmux new-session -s routing_session 'python3 ~/FYP/route.py
    eth2 eth3';
```

After detaching the session, we can then start another session and execute Snort:

```
> tmux new-session -s snort_session 'snort -c /etc/snort/snort.
    conf -i eth2 -de -l /var/log/snort -A full;'
```

We also use the following options in Snort:

- `-c` specifies our config file
- `-i` specifies interface
- `-d` dump application layer
- `-e` show second layer header info
- `-c` specify log directory
- `-A` alert mode




(a) Snort Container eth2 traffic (b) Suricata Container eth3 traffic

Figure 5.6: Firewall Routing

After processing all the packets, we can go to `/var/log/snort` to view all the alerts, see Fig. 5.7.

## 5.5 Suricata

The Suricata setup is quite similar to Snort but we do not need to run the routing program since it's our final destination in the service chain. We use `apt install suricata -y` to install Suricata and it comes with default rules activated, which is suitable for our needs. Next, we must change the following line in the `/etc/suricata/suricata.yaml` file to allow traffic from networks other than our own.

```
HOME_NET: "any"
```

We start Suricata using `tmux` with the following command:

```
> tmux new-session -s suricata_session 'suricata -c /etc/
    suricata/suricata.yaml -l /var/log/suricata -i eth3'
```

After processing all of the packets, we can examine all of the alerts by checking `/var/log/suricata/fast.log`, as shown in Fig. 5.8.

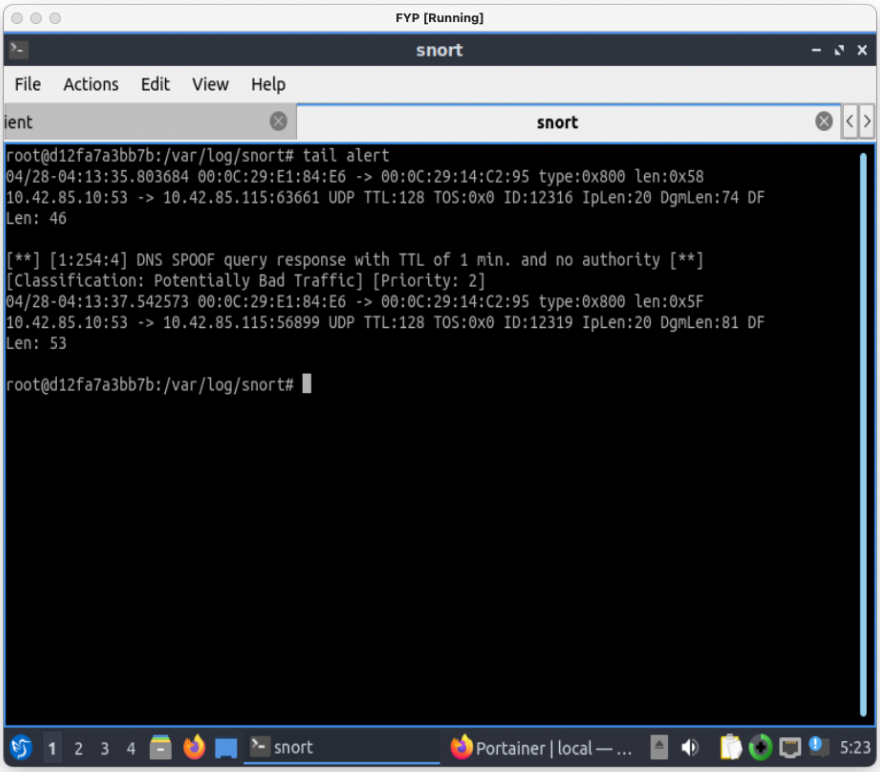



Figure 5.7: Snort Alerts

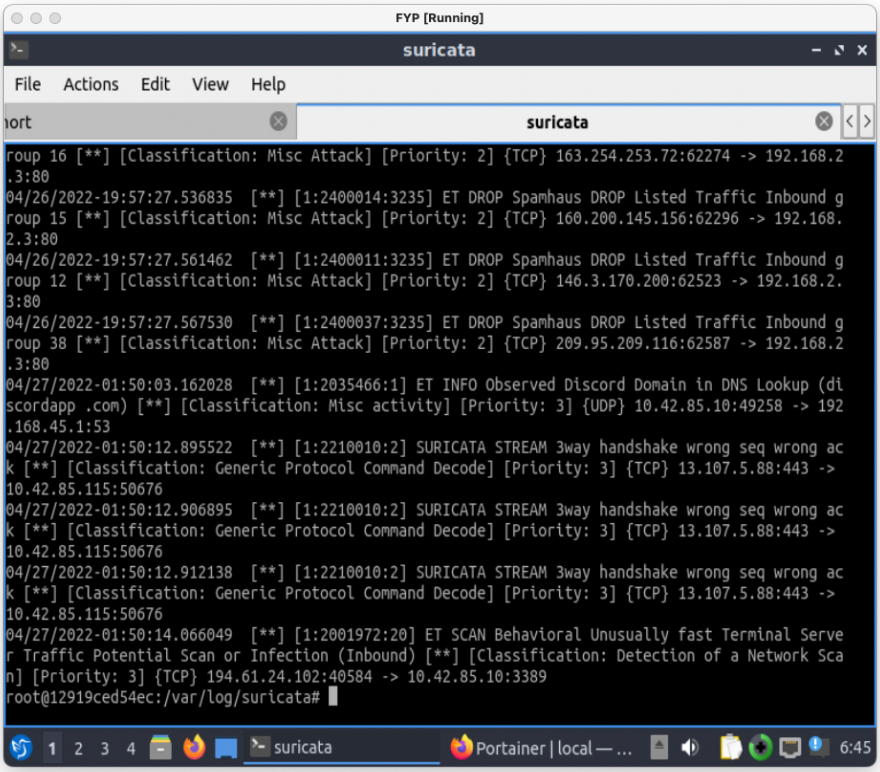



Figure 5.8: Suricata Alerts

# Chapter 6: Project Workplan

Below is the planning of the project drafted using a Gantt Chart. The implementation of SFC from week 1 to week 4 is crucial as it provides a foundation to build on top on in later weeks.

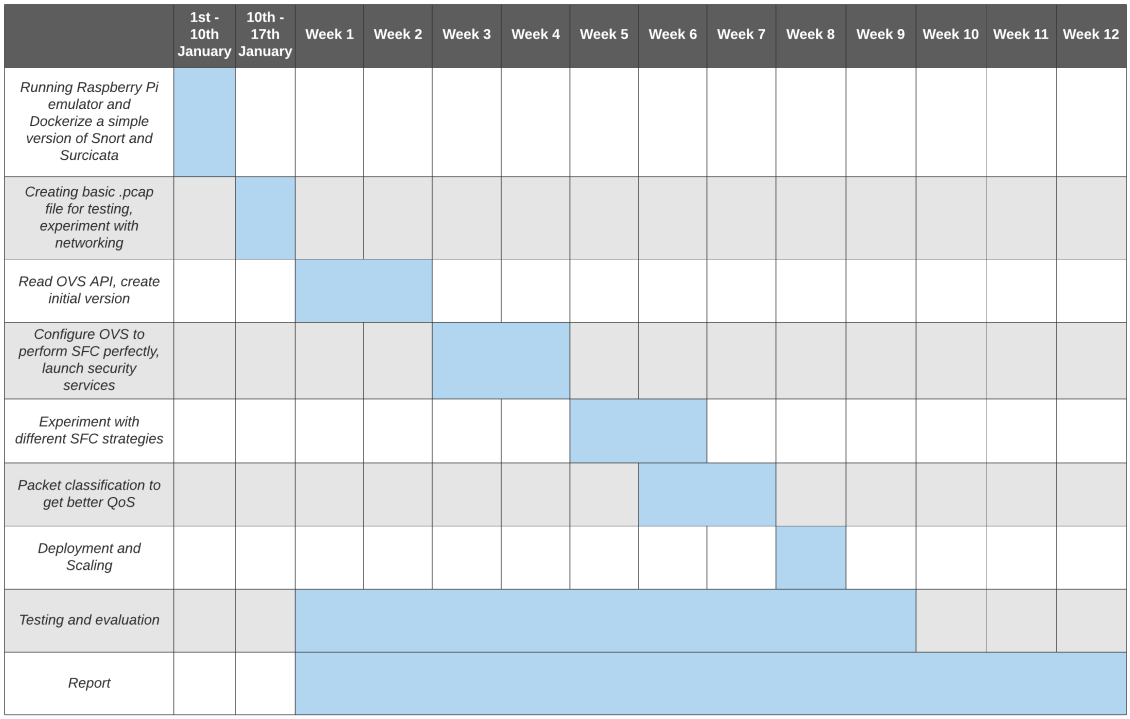



Figure 6.1: Project Gantt Chart

There are a few planned milestones for this project which are listed below:

- Develop and containerise security services (Snort, Suricata, etc)
- Implement SFC using Open vSwitch
- Implement SFC strategy and evaluated it's performance
- Implement packet classification to improve performance
- Deploy final service on a Raspberry Pi and evaluated performance

If the milestones have been completed earlier in advance, the following features will also be looked into:

- Multiple device support
- Native support of integrated Wifi chip
- Combining multiple edge nodes (Rapsberry Pis) to work together by sharing information
- Orchestration of multiple edge nodes using Kubernetes/K3s

# Chapter 7: Results and Evaluation

## 7.1 Introduction

Overall, the following goals has been achieved in my implementation:

- Developed and containerized security services (Snort, Suricata, etc)
- Implement SFC using Open vSwitch
- Implemented SFC and evaluated it's performance
- Implemented packet classification to improve performance

Due to supply chain issues, I was not able to obtain a Raspberry Pi device from the school, so I had to implement the SFC chain on a VM. However, we should be able to easily deploy the SFC implementation on a Raspberry Pi running Ubuntu.

## 7.2 Testing Environment

To emulate a Raspberry Pi, a virtual machine is created on a laptop with the specifications detailed in Table 7.1 and it is used to evaluate the performance of our implementation. The amount of RAM provided to the VM corresponds to that of a normal Raspberry Pi, allowing us to gain a more realistic picture of real-world performance. Besides, we are using the Lubuntu image for our VM since it is faster and a lightweight alternative to the Ubuntu desktop distribution. To clarify, this choice is based on the fact that we do not need to utilize the majority of the tools available in the Ubuntu Desktop image, and Lubuntu would likely run better in resource-restricted circumstances due to fewer background processes.

| | **Virtual Machine** |
|---|---|
| CPU | 1.4 GHz Quad-Core Intel Core i5 |
| RAM | 4GB DDR4 |
| OS | Lubuntu 22.04 |

Table 7.1: Machine Specifications

### 7.2.1 PCAP Files

Two PCAP files generated by different malware has been used and is listed below:

- Aleta Ransomware pcap - `aleta.pcap` [31]
- WannaCry ransomeware pcap - `wannacry.pcap` [32]

### 7.2.2 IDPS Rules

IDS such as Snort and Suricata act as network packet sniffers, initiating actions and recording events and information linked to them in a log file and/or database based on comparisons of packet contents with known viral signatures encapsulated as rules [33]. We will be using Snort v2.19.0 with the registered ruleset, which contains approximately 10096 detection rules. For Suricata 6.0.5, we will be using the latest updated rule set obtained using `suricata-update` which contains 33248 rules in total. By using a large rule set for both IDS, we can get a better sense of how our system will perform in real-world conditions.

## 7.3 Testing Scenarios

Multiple experiments have been set up to get a better sense of how our implementation performs and scales.

### 7.3.1 Basic Test

First, we start by measuring the number of alerts, packet processing rate, and drop rate for both Snort and Suricata in the SFC architecture. The pcap file used in this test is `wannacry.pcap`, and it is replayed at 3 Mbps, as shown in Table 7.2.

| | **Snort** | **Suricata** |
|---|---|---|
| Packets Processed | 11336 | 10398 |
| Dropped Packets | 0 | 0 |
| Alerts | 3634 | 1513 |
| Processing Rate (Packets/sec) | 50.6 | 46.4 |

Table 7.2: Basic Test with wannacry.pcap

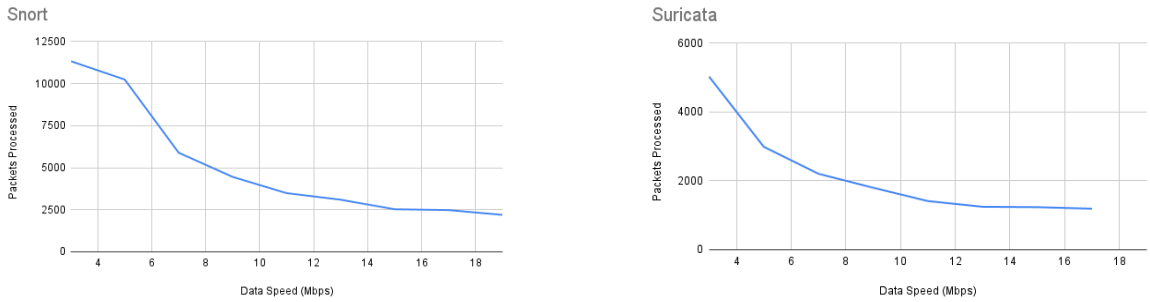
It's evident that, with this configuration, both Snort and Suricata can process packets quickly and have large alert counts. Since we are replaying packets at a slower speed, both services did not experience any dropped packets. Some packets have been lost between Snort and Suricata, likely due to packets being dropped at the kernel level.

### 7.3.2 Large PCAP Test

With the larger `aleta.pcap` file, we try to measure the performance of both containers. This helps us understand how the system will perform under longer duration and higher packet rates. When repeating packets at 10Mbps for 587 seconds, the virtual machine experiences performance difficulties and becomes unresponsive occasionally. This is likely due to higher CPU and RAM consumption compared to the smaller `wannacry.pcap` file. In Table 7.3, we can see that although the number of packets processed increased, the processing rates of both Snort and Suricata decreased by a significant amount.

| | Snort | Suricata |
|---|---|---|
| Packets Processed | 22669 | 9684 |
| Dropped Packets | 0 | 0 |
| Alerts | 27 | 5 |
| Processing Rate (Packets/sec) | 38.61 | 16.5 |

Table 7.3: Large PCAP Test with aleta.pcap




(a) Snort Packets Processed

(b) Suricata Packets Processed

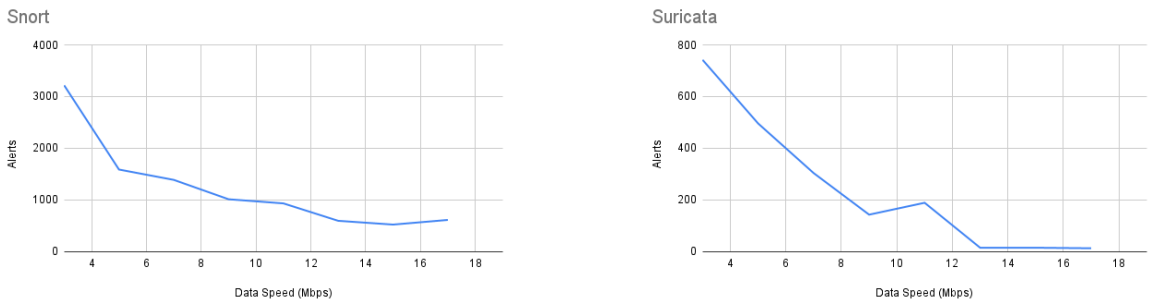
Figure 7.1: Packets Processed

### 7.3.3 Packets Processed

Next, we try to replay packets from 3 to 19 (with intervals of 2) Mbps using the `wannacry.pcap` file and try to observe the efficiency of our implementation. From Fig. 7.1, we can see that as the data speed increases, the number of packets processed by Snort and Suricata drastically reduces. When the data speed reaches 19 Mbps, the number of packets processed is five times lower than the initial value. This is likely because Snort and Suricata are starting to drop packets since we are running on a resource-constrained device.

### 7.3.4 Number of Alerts

Similar to the number of packets processed, we noticed that when we increase the packet replay speed from 3 to 19 (with intervals of 2) Mbps using the `wannacry.pcap` file, the number of alerts also decreases, see Fig 7.2.




(a) Snort Alerts

(b) Suricata Alerts

Figure 7.2: Number of Alerts

This makes a lot of sense since fewer packets will be analyzed by Snort and Suricata, resulting in fewer alerts.

## 7.4 Overall Evaluation

All of the tests show that the current is functioning properly and allowing packets to reach the Snort and Suricata containers in an anticipated manner. It is evident that this implementation performs well under low data rate speeds, and both Snort and Suricata were able to achieve a high packet processing rate and an alert rate. However, when exposed to high data speed environments, our system suffers from performance issues and fewer packets get processed, hence fewer alerts are observed as well. This is largely due to the high CPU and memory consumption of Snort and Suricata, which affects performance in resource-constrained environments. By analysing 7.1 and 7.2 we can conclude that the ideal range of network traffic is within the 1 to 5 Mbps range where we can still get relatively high packet processing rate and alert rate.

# Chapter 8: Future Work

## 8.1 Docker Compose

A potential improvement to the current implementation is to use `docker-compose` and Dockerfiles to spin up all the containers automatically without much user interaction. Bash scripts are then executed to set up the OVS bridges and connect the containers. Dockerfiles are especially useful since we can specify commands to install packages and libraries when we first set up the containers. Furthermore, this will simplify the deployment of the SFC implementation to other devices significantly, which saves us time.

## 8.2 Live Traffic Monitoring

Since I didn't get the chance to deploy an actual Raspberry Pi, a future improvement is to deploy it to a resource-constrained node and monitor live traffic with it. For instance, in Fig. 8.1, we try to monitor traffic through the Raspberry Pi Network Card directly and set up a Wifi access point for other devices. In other words, the Raspberry Pi will act as a man-in-the-middle and filter out malicious traffic or detect attacks before reaching the end-users.

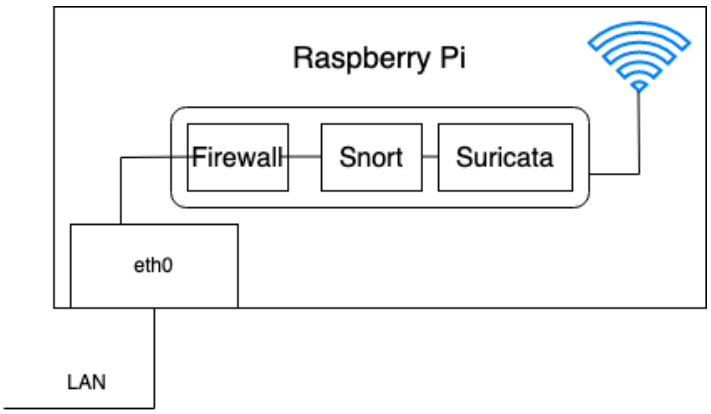



Figure 8.1: Live Traffic Monitoring

## 8.3 Orchestration of multiple edge nodes using Kubernetes

If live traffic monitoring is successfully implemented, the orchestration of multiple Raspberry Pis can also be looked into. By linking the SFC enabled Raspberry Pis together, important information such as security alerts and packet processing can be shared between the devices. For instance, if one

of the Raspberry Pis detects an intrusion alert, it can quickly send an alert to the other Raspberry Pis so that other clients can be alerted of this security breach. The orchestration and linking of multiple edge nodes is a non-trivial task, so the proposed approach is to use a orchestration service such as Kubernetes to set up this architecture.

# Chapter 9: **Summary and Conclusions**

The goal of this study was to provide an SFC implementation for security services on a resource-constrained node. SFC is a concept developed to allow for the provision of dynamic and flexible services in softwarised networks. Service function chains are a sequence of multiple virtual network functions (VNFs) that enable traffic steering. For this project, our VNFs will be security services such as Snort and Suricata. Based on quantitative and performance analysis, we can also conclude that SFC is a good choice candidate for securing edge networks as it allows us to deploy resource-constrained nodes such as Raspberry Pis and Arduino chips closer to the end-users. Not only does this approach save costs significantly, but we are also able to reduce the latency and response time experienced by users. Aside from security use, this implementation can potentially be used for HTTP video optimization and QoS provisioning.

SFC was achieved in this research by establishing OVS bridges among Docker containers, which allow for packets to pass through sequentially. The dynamic nature of service function chains provides the flexibility of adding new chains and rerouting packets dynamically, which can be used to adapt to different usage requirements. By conducting different tests and experiments, we can also see that the SFC implementation performs well at lower network speeds (1 - 5 Mbps). When exposed to high data speed environments, however, fewer packets get processed and hence fewer alerts are observed. This is largely due to the high CPU and memory consumption of Snort and Suricata.

# Acknowledgements

I'd want to express my gratitude to my supervisor, Prof Madhusanka Liyanage, for all of his assistance and advice with my thesis. I'd also like to thank Pasika Ranaweera, without whom none of this would have been possible. I'm also grateful for the help I've gotten from the rest of my family. Finally, I'd want to express my gratitude to the university for providing me with the chance to conduct my thesis.

# Bibliography


1. *Netresec public PCAP files* https://www.netresec.com/?page=PcapFiles. Accessed: 2021-12-03.
2. Kaur, K., Mangat, V. & Kumar, K. A comprehensive survey of service function chain provisioning approaches in SDN and NFV architecture. *Computer Science Review* **38,** 100298. ISSN: 1574-0137. https://www.sciencedirect.com/science/article/pii/S1574013720303981 (2020).
3. Bh, D., Jain, R., Samaka, M. & Erbad, A. A Survey on Service Function Chaining. *Journal of Network and Computer Applications* **75** (Sept. 2016).
4. Liyanage, M., Imrith, V. N., Ranaweera, P. & Damree, S. Enabling Fog Computing based Dynamic Security Service Function Chaining for 5G IoT (2021).
5. Zhang, D. *et al.* P4SC: A High Performance and Flexible Framework for Service Function Chain. *IEEE Access* **PP,** 1–1 (Oct. 2019).
6. Dong, Y., Bai, J. & Chen, X. A Review of Edge Computing Nodes based on the Internet of Things, 313–320 (Jan. 2020).
7. Perez, J., Gutierrez-Torre, A., Berral, J. & Carrera, D. A resilient and distributed near real-time traffic forecasting application for Fog computing environments. *Future Generation Computer Systems* **87** (May 2018).
8. Sha, K., Yang, T. A., Wei, W. & Davari, S. A survey of edge computing-based designs for IoT security. *Digital Communications and Networks* **6,** 195–202. ISSN: 2352-8648. https://www.sciencedirect.com/science/article/pii/S2352864818303018 (2020).
9. Maksimovic, M., Vujovic, V., Davidović, N., Milosevic, V. & Perisic, B. *Raspberry Pi as Internet of Things hardware: Performances and Constraints* in (June 2014).
10. Nedu, V. & Megalapete Puttaraju, A. R. *A survey on Docker and its significance in cloud* in (Feb. 2016).
11. Khraisat, A., Gondal, I., Vamplew, P. & Kamruzzaman, J. Survey of intrusion detection systems: techniques, datasets and challenges. *Cybersecurity* **2,** 20. ISSN: 2523-3246. https://doi.org/10.1186/s42400-019-0038-7 (July 2019).
12. Sandhu, H. & Kaur, M. *Review paper on Snort and reviewing its applications in different fields* in (2017).
13. BOUDI, A., FARRIS, I., BAGAA, M. & TALEB, T. Assessing Lightweight Virtualization for Security-as-a-Service at the Network Edge. *IEICE Transactions on Communications* **E102.B,** 970–977. ISSN: 0916-8516 (2019).
14. Sforzin, A., Marmol, F. G., Conti, M. & Bohli, J.-M. RPiDS: Raspberry Pi IDS — A Fruitful Intrusion Detection System for IoT. *2016 Intl IEEE Conferences on Ubiquitous Intelligence & Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People, and Smart World Congress (UIC/ATC/ScalCom/CBDCom/IoP/SmartWorld),* 440–448 (2016).
15. Baronti, P. *et al.* Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards. *Computer Communications* **30.** Wired/Wireless Internet Communications, 1655–1695. ISSN: 0140-3664. https://www.sciencedirect.com/science/article/pii/S0140366406004749 (2007).
16. Imrith, V. N., Ranaweera, P., Jugurnauth, R. A. & Liyanage, M. Dynamic Orchestration of Security Services at Fog Nodes for 5G IoT. *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)* **00,** 1–6 (2020).

17. Bonafiglia, R., Cerrato, I., Ciaccia, F., Nemirovsky, M. & Risso, F. Assessing the Performance of Virtualization Technologies for NFV: A Preliminary Benchmarking. *2015 Fourth European Workshop on Software Defined Networks,* 67–72 (2015).

18. *FYP Gitlab Repo* https://csgitlab.ucd.ie/jasontcg/FYP. Accessed: 2022-04-27.

19. Qadir, Q. M., Kist, A. A. & Zhang, Z. Mechanisms for QoE optimisation of Video Traffic: A review paper. *CoRR* **abs/2008.09156.** arXiv: 2008.09156. https://arxiv.org/abs/2008.09156 (2020).

20. Yu, R., Xue, G. & Zhang, X. QoS-Aware and Reliable Traffic Steering for Service Function Chaining in Mobile Networks. *IEEE Journal on Selected Areas in Communications* **35,** 2522–2531 (2017).

21. *Open vSwitch Documentation* https://docs.openvswitch.org/en/latest/. Accessed: 2022-04-27.

22. *Play With Container Network Interface* https://arthurchiao.art/blog/play-with-container-network-if/. Accessed: 2022-04-27.

23. Torino, D. & Trani, R. *Integrating VNF Service Chains in Kubernetes Cluster* in (2020).

24. *Network Service Mesh Documentation* https://networkservicemesh.io/. Accessed: 2022-04-27.

25. Varis, N. *Anatomy of a Linux bridge* in (2012).

26. *Linux Bridge* https://developers.redhat.com/sites/default/files/styles/article_feature/public/blog/2018/10/bridge.png?itok=fic3r7Jv. Accessed: 2022-04-27.

27. *Portainer Installation Guide* https://docs.portainer.io/v/ce-2.9/start/install/server/docker/linux. Accessed: 2022-04-27.

28. *Suricata Documentation* https://suricata.readthedocs.io/en/latest/performance/packet-capture.html. Accessed: 2022-04-27.

29. Rountree, D. in *Security for Microsoft Windows System Administrators* (ed Rountree, D.) 71–107 (Syngress, Boston, 2011). ISBN: 978-1-59749-594-3. https://www.sciencedirect.com/science/article/pii/B978159749594300003X.

30. *Cannot start service using systemctl inside docker container* https://stackoverflow.com/questions/46800594/start-service-using-systemctl-inside-docker-container. Accessed: 2022-04-27.

31. *Aleta pcap file* https://www.malware-traffic-analysis.net/2017/05/18/index2.html. Accessed: 2022-04-27.

32. *WannaCry pcap file* http://dataset.tlm.unavarra.es/ransomware/. Accessed: 2022-04-27.

33. Kumar, V. Signature Based Intrusion Detection System Using SNORT. *International Journal of Computer Applications & Information Technology* **1,** 7 (Nov. 2012).