TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

DIPLOMA THESIS

Design and implementing network security primitives for core 5G infrastructure

Author: Stavros Lyronis

Committee:

Associate Professor Sotiris Ioannidis

Professor Apostolos Dollas

Professor Aggelos Bletsas

September 2022

Abstract

We live in an era where modern networking has become a constant battlefield between hackers and cyber-security operators. Malicious users launch attacks in any internet facing system that shows vulnerabilities. Due to the growth of the rate of these attacks, the former naive stationary approaches on computer security are being abandoned and the need for a new mobile way to defend is required. By taking advantage of Active Networks, the PRINCIPALS project introduces a novel architecture for safe programmability and adaptability in 5G networks. In this thesis we develop five network security primitives for core 5G infrastructure and some language security techniques that will be used in the PRINCIPALS project. These network security primitives will be addressing current and anticipated security problems in 5G networks, through custom-yet-adaptive logic expressed as code. This code, tested and evaluated, will be containerised and deployed inside the Active Management Environment (AME) that PRINCIPALS provides. The application of language security techniques that are also implemented in this thesis will set an upper-bound limit on what actions the code-carrying packets used in PRINCIPALS can take. In this way the flexibility and adaptability will be secured, while maintaining security in the 5G Network.

Acknowledgements

I would like to give a massive thank you to my supervisor, Dr.Sotiris Ioannidis, for all of the assistance he has given me throughout this process. I would also like to give a special thank you to Dr.Spiros Antonatos and MS.Grigorios Ntousakis for their support and excellent communication that made me come this far. All of them were always willing to give detailed explanations whenever I had an issue, and it is really appreciated. I would not be able to complete this thesis without their incredible aid. I would like to thank my incredible parents, Giannis and Despoina, for their love, compassion and support all these years, as well as my three wonderful brothers, Manos, Nikos and Vasilis. I wouldn't be here without you.

Contents

Chapter 1

Introduction

In today's era, mobile communications are an undisputed part of our modern life. During the last four decades, with the evolution of mobile wireless technology [14] starting from 1G in 1980 and reaching the 5G in 2019, we have had a number of new features and capabilities introduced to us in each mobile network generation. With the growth of them, the number of demands for using these new technologies also has gradually increased, and the mobile interaction has become a daily part of our lives, especially with the evolution of Internet of Things (IoT) [13].

Between every new generation of wireless technology, new challenges and opportunities have emerged. The fifth generation of wireless technology comes with some guarantees of acceleration of data transfer, through bigger channels, less latency, the ability of multiple devices to be linked at once, etc. However, in every dramatic advancement of a new technology, there are always security issues that we must bear in mind [2], [31]. In 2019 global operators started launching 5G networks that enabled the movement and access of vastly higher quantities of data. This resulted in a much wider surface for the attackers to make their moves. It's safe to say that the providers of the infrastructure surface must be very cautious and take the necessary steps to prevent the attackers from having the ability to make a move or in case an attack is realized, to take rapidly and with precision the right actions to neutralize it. Cybersecurity operators have the duty to be one step ahead of the attackers and find the latest tactics to deal with them.

1.1 Aim of the thesis

5G networks interconnect billions of devices and offer high bandwidth and low latency guarantees. In this environment, we want to have flexible cybersecurity defenses that can be deployed quickly in a variety of settings, in order to counter the inevitable number of attacks that are coming. The PRINCIPALS project aims to be able to counter such attacks rapidly and with precision. The goal of this thesis is the design and the implementation of security primitives that are going to be used in the PRINCIPALS project in order to achieve its goal. These network security primitives will be deployed inside the PRINCIPALS environment and will provide a plethora of different defensive cyber operations to counter any possible attacks in the 5G infrastructure. Primitives will use containerised code that is tested and evaluated.

The application of some language security techniques will accompany these network security primitives to secure a safe and active management environment. These language security techniques will provide the necessary extraction of permissions that the *mobile* defensive cyber operations (DCOs) require. These *mobile* DCOs will guarantee flexibility and adaptability in the 5G network.

1.2 Thesis contribution

It has already been mentioned that PRINCIPALS is a project that will provide a novel architecture for safe programmability and adaptability in 5G networks. It will enable more secure networks, since it provides the ability of *mobile* defensive cyber operations. The 5G Network uses Kubernetes as the infrastructure needed for scaling. The PRINCIPALS architecture is a virtualised architecture built on top of Kubernetes as well. The following summarizes this thesis' contribution to the PRINCIPALS project:

- Five network security primitives, that PRINCIPALS will use as defensive mechanisms are designed and implemented. These primitives, also known as FAMElets, are realized in the Kubernetes environment and are full-fledged code that provide complex, performance-sensitive, and possibly privileged functionality (FAMElets will be explained thoroughly below). FAMElets will be deployed inside each node and conduct defensive cyber operations for current and anticipated network security issues in the 5G network. These five primitives are:

  1. Snort, a Network Intrusion Detection&Prevention System.
  2. DNS Sinkholing, a mechanism used to protect users from accessing malicious or unwanted domains.
  3. Flow Recording, a tool used to monitor, analyze and store information about traffic from specific domains.
  4. DDoS Detector, a tool used to detect possible Distributed Denial of Service attacks.
  5. DGA Detector, a tool used for detection of Domain Generation Algorithm, a technique used for a cyber attack by creating multiple domains from an algorithm, that generate traffic.

These network security primitives can provide a number of different mechanisms and functionality just by changing the action to the cybersecurity operator's desire. This can be implemented by passing different commands to the bridge, as you will see below. With the deployment of these primitives, each 5G/Kubernetes node will have some security guarantees about a plethora of different cyber attacks.

- Every single primitive, from the five mentioned above, was rewritten in Javascript. From the smallest script to the largest, everything used in these primitives was written again from scratch in Javascript. Besides the new scripts, this means that all of the containers had to be created again. New Dockerfiles, new docker images, new archive in the docker repository. The same applies for the yaml files and the pods in the Kubernetes environment. The whole process of how a primitive is realized will be explained thoroughly in Chapter 3. This implementation in Javascript was necessary in order to apply the code analysis tools.

- A code analysis was also implemented, as a part of this thesis. Code analysis, static and dynamic, was conducted for every script in every primitive implemented. This analysis required two tools, MIR and LYA, for static and dynamic analysis respectively. MIR and LYA both are tools that conduct code analysis for Javascript implemented scripts. This language security technique provides the necessary generation of permissions TAMElets (will be explained thoroughly below) require. TAMElets will rely on FAMElets for much of their functionality and will provide a dynamic configuration on the 5G Network. The set of permissions MIR, the static analysis tool will generate will act as an upper bound limitation in what actions each TAMElet can take.

- An evaluation of the DDoS Detector primitive is performed as the last part of this thesis' contributions. The testing focuses particularly on the different variants of the Mirai botnet attacks. Each of the three different case scenarios of DDoS attacks (UDP, HTTP, SYN) is tested and evaluated.

1.3 Thesis overview

This thesis' contributions are presented in Chapter 3 and Chapter 4 along with an evaluation in Chapter 5, with the main focus being the deployment of our network security primitives in the Kubernetes environment. However, in order to understand the way each primitive works and why they are designed and implemented the way they are, you should have a background knowledge.

Chapter 2 provides the necessary background knowledge you need. Multiple concepts like Virtual Machines, Containers, Docker, etc are explained. Also a very detailed explanation of the 5G Network is given, the latest generation of network that will become a surface of cyber wars. At last it provides a short review of the PRINCIPALS Architecture. The work of this thesis is going to be used inside the PRINCIPALS framework as the environment to execute. This novel architecture will provide safe programmability and adaptability in 5G networks and will enable more secure networks and endpoints relatively to the current state of the art.

Chapter 3 provides a detailed explanation of the Network Security Primitives developed, the main focus of this thesis. These primitives are going to be used by the PRINCIPALS project to conduct Defensive Cyber Operations, in order to defend upon the growth of cyber attacks in the 5G network. The primitives, or FAMElets as they are mentioned inside the PRINCIPALS project, are five in total and they are logic expressed as containerised code, deployed in the Kubernetes environment for the purpose of defending against known and anticipated cyber problems.

Chapter 4 provides a detailed explanation of the language security techniques implemented in this thesis. An explanation of what code analysis is and how it is used is also provided, as well as a synopsis of the tools used for both dynamic and static analysis. Beside the explanation of the tools themselves, we show why and how we used them inside the project and the work needed to be done to use them.

Chapter 5 provides the last part of this thesis' contribution, the evaluation of the DDoS Detector primitive. A description about the metrics, the process, and the critical questions that need to be answered is also provided, as well as the results of the three possible cases of the DDoS attack (UDP flooding, HTTP flooding, SYN flooding).

Chapter 6 provides the Related work that has already done. This previous work inspired, or even better enabled us to think and realize PRINCIPALS. PRINCIPALS will be built upon the work done in this previous projects.

Chapter 7 shows the conclusions drown from this thesis.

Appendix 1: In this appendix it is shown a step by step procedure of how to build your own containerised app from scratch, with the help of Docker. As a practical example, the Dockerfile of one of the primitives implemented in this thesis is used, in order to get an actual experience of a container.

Appendix 2: In this appendix it is shown how to use Python's Scapy as a tool to monitor the network traffic. As a practical example, the script of one of the primitives is presented, in order to get an actual experience of how to use Scapy's sniffing.

Chapter 2

Background

Some background knowledge is mandatory for someone to understand the idea behind and the implementation of this thesis. That is why in this chapter some concepts like Virtual Machines, Container, Docker, Kubernetes, etc are being explained. The implementation of this thesis depends on these elements since every single network security primitive uses most, if not all of them. Here is what is going to be needed.

2.1 Virtual Machines

A virtual machine or a VM is a virtualized instance of a computer system [32]. Typically a virtual machine is launched from an image that contains an Operating System and all of the libraries and binaries that come with a standard version of it. The nature of the virtualization means that each virtual machine is isolated from each other and can only interact with the host machine and hardware peripherals. This interaction is done via the hypervisor. The hypervisor is also known as a virtual machine monitor or VMM [37]. It is software that creates and runs virtual machines. A hypervisor allows one host computer to support multiple guest VMs by virtually sharing its resources, such as memory and processing. Hypervisors make it possible to use more of a system's available resources and provide greater IT mobility since the guest VMs are independent of the host hardware. The amount of computing resources that allocated in a VM can vary. That's what makes them a good choice for cloud computing tasks since the service provider can offer its users the ability to pay only for the resources they want to use. Several VMs can be hosted on the same infrastructure, with customisable specifications. The hypervisor starts and stops these VMs and acts as an intermediate between the host Operating System and the guests. The Host OS then has direct access to the underlying Infrastructure.

2.1.1 Virtual Machine Security

One of the benefits in using virtual machines is that each instance is isolated from every other instance. It has its own storage, Operating System, libraries and system binaries. This allows multiple guests to run on the same infrastructure without one guest being able to access another or corrupt its host. This is the principle by which virtual machines can securely enable the concept of a platform as a service. However, the increased adoption of virtualization has also led to increased concerns about the security risks associated with virtualization [30].

2.2 Containers

A container is a virtual, isolated computing environment, similar to virtual machines [33]. The programs that run within a containerised environment will believe that they are running on a normal computer. Containers are virtualized at the Operating System level, unlike VMs. This means that containers running on the same host share the same kernel while being unaware of the other containers running on the same host. Containers are bundled with any necessary libraries or any other dependencies their applications require. This allows an easy deployment without worrying about whether the target environment can run the client application or not. Container images themselves can be built upon other container images [34]. That's what it makes it so easy for a developer to extend an existing container to suit their needs. This new container can be pushed and anyone will now be able to use it as a base to extend upon themselves. There are lots of common benefits of running your applications in containers and in Virtual Machines [8], [9]. Programs can be run in isolated, from their host, environment and each other. An additional benefit is that they are more lightweight, in other words the amount of memory that they require is smaller than an equivalent virtual machine, as they only contain their applications and the necessary libraries to run them. Containers don't have to handle any of the kernel space tasks, nor do they need to include them. This reduces the amount of resources utilized. Moreover, multiple containers can share data through a shared file system if they want to. Containerisation offers improved portability of applications across different underlying operating systems and cloud infrastructure. The container engine manages the starting and stopping of the containers. All of the containers share access to the Host OS's kernel for kernel level tasks.

2.2.1 Container Security

The fact that containers are considerably more lightweight than VMs means that there is less of an issue in winding down an infected container. It can be dealt by launching a new instance, much easier than there would be with a VM. A potential drawback to choosing a container over a virtual machine is the matter of security. Containers have access to the shared kernel, so you can say they are less *removed* from the host OS than VMs are [7], [38].

2.3 Docker

Docker [10] is the software that commoditized containers and made them popular, by offering advanced capabilities with a familiar user interface that is pretty easy to use. Docker builds on two kernel features:

- Namespaces: Namespaces are a feature of the Linux kernel that partitions kernel resources such that each set of processes can see a different set of resources. Some examples of resources are pid, ipc, network etc.
- Cgroups: Cgroups are also a feature of the Linux kernel that limit the usage of certain resources for specified processes. For example, by using cgroups a user can limit a process' CPU and Memory allocation, network and I/O bandwidth.

Those two features work together to give a flexible and isolated environment for a process to run. Docker builds on that and adds the concept of images.

2.3.1 Docker Images

A Docker Image is a file usually containing a program (eg Cassandra, MySQL) with all its dependencies. Using a Docker image, a user can launch a Container from it. Essentially, Docker took the concept of images from the world of Virtual Machines and applied it to containers. A Docker Image is made of several read-only layers that are stacked on top of each other, using a union-capable file system. This way, common layers are reused between different images and a Copy-On-Write strategy is applied.

2.3.2 Image Registries

After Docker introduced us with the concept of an image, it took a step forward by adding sharing and versioning capabilities. This was done by creating an image registry, Docker Hub [11]. Docker Hub can be described as an image version of what Github[15] is used by developers to version code. An image is versioned using tags, which are pretty much like git branches. For example, the Ubuntu image can be found with the Xenial tag or the bionic tag. This way someone can use an already existed image from the Docker Hub and use it to create an new image of his own.

2.4 Container Orchestration

Managing large groups of containers in a deployment has many difficulties and complexities. Container orchestration tools were created to help with this process. Orchestration refers to the automation of key features of the management of containerised systems. These are:

- Deploying new containers depending on configuration files. Containers do not have to be manually started. Instead they can be automatically started from a script. In this way, multiple interdependent containers can be launched at the same time.
- Coordinating deployed containers. Containers can be categorised and divided into groups based on whether they require each other's services or not.
- Network supplying for containers, that enables them to access or be accessed via a network. This can include access over the Internet as well.
- Flexible scaling of resources to meet demand. If a large number of users attempted to access a containerised system, the system load could exceed a certain threshold that would lead to a slow down. Creating new groups of containers could counter this problem.
- Availability and fault tolerance, achieved by adding some form of redundancy to the system. One container going down should not bring the whole system down.
- Stopping container instances when they are no longer needed. This also includes restarting containers that have entered a status of error or have been corrupted.

2.5 Kubernetes

Kubernetes [17] is an open source container orchestration tool. It was originally designed by Google engineers to manage the automation of container deployments. It can be run on a physical machine, cloud infrastructure or some kind of a hybrid scheme. In Kubernetes, the smallest deployable instance is called a pod, a single application instance. Kubernetes can handle systems consisting of multiple pods as we'll analyze further down below.

2.5.1 Kubernetes Design

Kubernetes design can be divided into the control plane and the nodes. The control plane contains the services which manage the nodes and the pods within the nodes. These services are:

- etcd: a service that Kubernetes uses to store all of its data, its configuration data, its state and its metadata. Kubernetes is a distributed system, so there is a need for a distributed data store like etcd as well. etcd lets all of the nodes in the Kubernetes cluster read and write data.
- kube-scheduler: a control plane process whose job is to assign Pods to Nodes. Depending to the constraints and the available resources, the scheduler determines which Nodes are valid placements for each Pod. Multiple different schedulers may be used within a cluster.
- kube-controller-manager: is a daemon that embeds the core control loops shipped with Kubernetes. The controller roll is to watch the shared state of the cluster through the api-server and make changes to move the current state towards the desired state. Examples of controllers that ship with Kubernetes today are the replication controller, endpoints controller, namespace controller, and service-accounts controller.
- cloud-controller-manager: is a Kubernetes control plane component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster. It only runs controllers that are specific to your cloud provider. If you are running Kubernetes on your own premises the Cluster won't have a cloud controller manager.
- kube-api-server: is a component of the Kubernetes control plane that is designed to scale horizontally, in other words, it scales by deploying more instances. You can run several instances of kube-apiserver and balance traffic between those instances.

Nodes can run on physical or virtual machines. They handle and control the creation and management of pods. Every cluster has at least one node. They consist of the following components:

- kubelet: an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of pod's specs, that are provided through various mechanisms and ensures that the containers described in those specs are healthy and running. The kubelet doesn't manage containers which weren't created by Kubernetes.

- kube-proxy: a network proxy that runs on every node of the cluster, implementing part of the Kubernetes Service concept. kube-proxy maintains network rules on nodes. These network rules allow external network communication to Pods. Kube-proxy uses the operating system's packet filtering layer, if there is one and it's available. Otherwise, it forwards the traffic itself.
- Container runtime: is the software that is responsible for running containers.

2.5.2 Pods

Pods are the smallest deployable units of computing that you can create and manage in Kubernetes. They represent the basic executable instance of an application in Kubernetes. They are deployed onto the worker nodes. A pod consists of one or more containers, their underlying storage and an IP address and it has the specification for how to run the container. All containers in the same pod are assigned the same IP address, and can communicate with each other through their shared data storage or via localhost.

2.6 Antrea

Antrea [4], designed by VMware2, is a CNI that operates at Layer 3/4, the Network/Transport layers. It provides networking and security services for a Kubernetes cluster. Antrea takes advantage of Open vSwitch [24] to implement Pod features of networking and security. In a Kubernetes cluster, Antrea creates two main objects:

- A Deployment (the Antrea Controller), that watches the NetworkPolicy, Pod, and Namespace resources from the Kubernetes API and it processes several components (Pod Selectors, Namespace Selectors, ip Blocks) notifying only interested nodes.
- A DaemonSet, that executes two containers (Antrea Agent and OVS daemons) on every Node. The DaemonSet also includes an Init container that installs the CNI plugin (Antrea-CNI) on the Node and, in general, ensures that the OVS kernel module is loaded and it is linked with the portmap CNI plugin. The Antrea Agent (deployed on each node) receives only the computed policies which affect Pods running locally on its Node. It directly uses the IP addresses, computed by the Controller, to create OVS flows enforcing the specified NetworkPolicies. In this way, only the nodes involved are notified.

When installed, the network provider creates on each node an OVS bridge with several interfaces used in different kinds of communications. Antrea uses encapsulation and the OVS bridge directly forwards packets between two local Pods. Indeed, for inter node communication, packets will be first forwarded to the tun0 port. Then, they will be encapsulated and sent to the destination Node through the tunnel. After that they will be decapsulated and injected through the tun0 port to the OVS bridge, and finally forwarded to the destination Pod.

2.7 Keras

Keras [16] is an open source software library that provides a Python interface for artificial neural networks. Keras is the high-level API of TensorFlow 2, that is designed for human beings, not machines. It runs on top of TensorFlow and since its built in Python, it is very user-friendly. It is an approachable, highly productive interface for solving machine learning problems. It mainly focuses on modern deep learning. Keras provides essential abstractions and building blocks for developing and shipping machine learning solutions with high iteration velocity. Keras follows the best practices for reducing cognitive load. It offers consistent and simple APIs, it minimizes the number of user actions required for common use cases and it also provides clear and actionable error messages.

2.8 Multus

Multus [20] CNI is a container network interface (CNI) plugin for Kubernetes. It enables the attachment of multiple network interfaces to pods. Usually, a Kubernetes pod has two network interfaces, its primary and a loopback interface. Multus can create a multi-homed pod that has multiple interfaces. Multus acts as a *meta-plugin*, a CNI plugin that can call multiple other CNI plugins. Multus CNI follows the Kubernetes Network Custom Resource Definition De-facto Standard to provide a standardized method. Following this standard method, it specifies the configurations for additional network interfaces. This standard is put forward by the Kubernetes Network Plumbing Working Group. It provides a very useful installation guide that makes it easy to work with if you are new to it.

2.9 5G Network

The term *5G* is used to refer to the fifth generation of mobile wireless technologies, which originated with analog mobile telephony in the late 1980s and has progressed to the point where all people and things can be connected to the Internet. Every generation's of mobile technology aim is to provide connectivity anywhere and anytime. The underlying technological objectives and the capabilities of the network, however, have continued to shift into a new generation around every 10 years approximately. Each generation is designed to serve the needs of the current society over a duration of 2 to 3 decades around the globe. The impact of these generations on the way we communicate may be viewed across various dimensions, such as:

- Service offerings.
- Air-interfaces.
- Data rates.
- Spectrum ranges.
- Performance.

Early generations began with only isolated concerns, such as telephone conversations, while later ones evolved to digital data communication and more sophisticated service architectures. Accordingly, the focus in past generations was consistently on operation in wider spectrum ranges and with higher data rates and traffic handling capacities. These objectives continue to be important today, with 5G being able to operate over much wider range of frequency bands than ever before. The added attention to massive Internet of Things(IoT) support, the development of support for critical services, and the remaining of the core network functionality to better support distributed cloud computation and service orchestration expands the potential of 5G further.

The 5G system is built on radio access nodes and distributed and centralized data centers, allowing for a flexible allocation of workloads. These nodes and data centers are connected via programmable transport networks, which are connected via backbone nodes that carry the information from the access nodes to the data centers, where most of the data is stored and the network itself is managed. In addition to this, the management of applications, cloud, transport, and access resources can be allocated centrally in the data center or be flexibly allocated as necessary. 5G systems have a significant role to play, not just in the evolution of communications but in the evolution of businesses and society as a whole.

2.9.1 5G Network Architecture

5G is effectively a dynamic, coherent and flexible framework of multiple advanced technologies supporting a variety of applications. 5G utilizes a more intelligent architecture, with Radio Access Networks (RANs) no longer constrained by base station proximity or complex infrastructure. 5G leads the way towards distributed, flexible and virtual RAN with new interfaces creating additional data access points.

5G Architecture 3GPP : The 3rd Generation Partnership Project 3GPP [1] covers telecommunication technologies including RAN, core transport networks and service capabilities. 3GPP has provided complete system specifications for 5G network architecture which is much more service oriented than previous generations 3GPP. Services are provided via a common framework to network functions that are permitted to make use of these services. Modularity, reusability and self-containment of network functions are additional design considerations for a 5G network architecture described by the 3GPP specifications.

5G Spectrum and Frequency: Multiple frequency ranges are now being dedicated to 5G new radio (NR). The portion of the radio spectrum with frequencies between 30 GHz and 300 GHz is known as the millimeter wave, since wavelengths range from 1-10 mm. Frequencies between 24 GHz and 100 GHz are now being allocated to 5G in multiple regions worldwide. In addition to the millimeter wave, underutilized UHF frequencies between 300 MHz and 3 GHz are also being repurposed for 5G. The diversity of frequencies employed can be tailored to the unique applications considering the higher frequencies are characterized by higher bandwidth, through shorter range. The millimeter wave frequencies are ideal for densely populated areas, but ineffective for long distance communication. Within these high and lower frequency bands dedicated to 5G, each carrier has begun to carve out their own discrete individual portions of the 5G spectrum.

MEC: Multi-Access Edge Computing (MEC) [18] is an important element of 5G architecture. MEC is an evolution in cloud computing that brings the applications from centralized data centers to the network edge, and therefore closer to the end users and their devices. This essentially creates a shortcut in content delivery between the user and host, and the long network path that once separated them.

This technology is not exclusive to 5G but is certainly integral to its efficiency. Characteristics of the MEC include the low latency, high bandwidth and real time access to RAN information that distinguish 5G architecture from its predecessors. This convergence of the RAN and core networks will require operators to leverage new approaches to network testing and validation.

5G networks based on the 3GPP 5G specifications are an ideal environment for MEC deployment. The 5G specifications define the enablers for edge computing, allowing MEC and 5G to collaboratively route traffic. In addition to the latency and bandwidth benefits of the MEC architecture, the distribution of computing power will better enable the high volume of connected devices inherent to 5G deployment and the rise of the Internet of Things (IoT).

NFV: Network function virtualization (NFV) [36] decouples software from hardware by replacing various network functions such as firewalls, load balancers and routers with virtualized instances running as software. This eliminates the need to invest in many expensive hardware elements and can also accelerate installation times, thereby providing revenue generating services to the customer faster. NFV enables the 5G infrastructure by virtualizing appliances within the 5G network. This includes the network slicing technology that enables multiple virtual networks to run simultaneously. NFV can address other 5G challenges through virtualized computing, storage, and network resources that are customized based on the applications and customer segments.

5G RAN Architecture: The concept of NFV extends to the RAN through for example network disaggregation promoted by alliances such as O-RAN [23]. This enables flexibility and creates new opportunities for competition, provides open interfaces and open source development, ultimately to ease the deployment of new features and technology with scale. The O-RAN ALLIANCE objective is to allow multi-vendor deployment with off-the shelf hardware for the purposes of easier and faster inter-operability. Network dis-aggregation also allows components of the network to be virtualized, providing a means to scale and improve user experience as capacity grows. The benefits of virtualizing components of the RAN provide a means to be more cost effective from a hardware and software viewpoint especially for IoT applications where the number of devices is in the millions.

eCPRI: Network disaggregation with the functional split also brings other cost benefits particularly with the introduction of new interfaces such as eCPRI [12]. Radio Frequency (RF) interfaces are not cost effective when testing large numbers of 5G carriers as the RF costs rapidly increase. The introduction of eCPRI interfaces presents a more cost-effective solution as fewer interfaces can be used to test multiple 5G carriers. eCPRI is aimed to be a standardized interface for 5G used for instance in the O-RAN front haul interface such as the DU. CPRI in contrast to eCPRI was developed for 4G, however in many cases was vendor specific making it problematic for operators.

Network Slicing: Perhaps the key ingredient enabling the full potential of 5G architecture to be realized is network slicing. This technology adds an extra dimension to the NFV domain by allowing multiple logical networks to simultaneously run on top of a shared physical network infrastructure. This becomes integral to 5G architecture by creating end-to-end virtual networks that include both networking and storage functions. Operators can effectively manage diverse 5G use cases with differing throughput, latency and availability demands by partitioning network resources to multiple users or *tenants*.

Network slicing becomes extremely useful for applications like the IoT, where the number of users may be extremely high, but the overall bandwidth demand is low. Each 5G vertical will have its own requirements, so network slicing becomes an important design consideration for 5G network architecture. Costs, resource management and flexibility of network configurations can all be optimized with this level of customization now possible. In addition, network slicing enables expedited trials for potential new 5G services and quicker time-to-market.

Beamforming: Another breakthrough technology integral to the success of 5G is beamforming [5]. Conventional base stations have transmitted signals in multiple directions without regard to the position of targeted users or devices. Through the use of multiple-input, multiple-output (MIMO) arrays featuring dozens of small antennas combined in a single formation, signal processing algorithms can be used to determine the most efficient transmission path to each user while individual packets can be sent in multiple directions then choreographed to reach the end user in a predetermined sequence.

With 5G data transmission occupying the millimeter wave, free space propagation loss, proportional to the smaller antenna size, and diffraction loss, inherent to higher frequencies and lack of wall penetration, are significantly greater. On the other hand, the smaller antenna size also enables much larger arrays to occupy the same physical space. With each of these smaller antennas potentially reassigning beam direction several times per millisecond, massive beamforming to support the challenges of 5G bandwidth becomes more feasible. With a larger antenna density in the same physical space, narrower beams can be achieved with massive MIMO, thereby providing a means to achieve high throughput with more effective user tracking.

CNFs: While VNFs made a breakthrough by virtualising Network Functions, they still have limitations due to the *weight* of VMs, especially when it comes to scaling. Therefore, digital service providers moved toward delivering more agile services. They adopted a cloud-native approach, using both centralized and distributed locations for applications. This action benefited in matters of flexibility, scalability, reliability, and portability. Moving beyond virtualization to a fully cloud-native design helps push to a new level the efficiency and agility needed to rapidly deploy innovative, differentiated offers that markets and customers demand. That is when CNFs made their appearance. CNFs is the equivalent of VNF, but designed and implemented in containers. This containerization of network architecture components makes it possible to run a variety of services on the same cluster and more easily on-board already decomposed applications, while dynamically directing network traffic to the correct pods. The infrastructure 5G decided to use for the orchestration of these containers was Kubernetes.

2.10 PRINCIPALS

In order to further discuss the main topic of this thesis, the design and the implementation of the network security primitives, it is necessary to take a look at the PRINCIPALS project and its architecture. Nowdays, networks and the nodes they interconnect have become a battlefield. Malicious users have the freedom to use the network for all kind of malicious purposes like reconnaissance, origin obfuscation, propagation, attack and more. Over the years, a number of protection mechanisms have been developed and deployed that mostly take the form of a stationary, point defence. It is essential for a new mechanism that will allow the defensive side to conduct *mobile* defensive cyber operations (DCO). This new defensive mechanism will be able to match the speed, scale, and accuracy of attacker tools, without introducing new points of vulnerability or risk network instability. The PRINCIPALS project introduces a novel architecture that will provide safe programmability and adaptability in 5G networks. This architecture will enable more secure networks and endpoints compared to the stationary point of defense.

The PRINCIPALS core architecture relies upon domain-specific languages and restricted execution runtimes, along with suitable cryptographic protocols and authorization credentials. It runs atop locked-down open-source platforms that offer the core services necessary to implement a variety of security primitives. PRINCIPALS will thereby offer an active management environment (AME) that we will be able to quickly prototype. This prototype will be used to investigate, develop, and evaluate algorithms and approaches to addressing current and anticipated security problems through custom-yet-adaptive logic expressed as code (*AMElet*). AMElets will be deployed on-demand inside the network, possibly at scale, while also creating new defensive opportunities. *Thin* AMElets (TAMElets) are implemented in a novel domain-specific language (DSL) that offers inherent safety and security guarantees. TAMElets compose and supervise fundamental security and data analytics services built into the 5G infrastructure. These services can themselves be extended via persistent *fat* AMElets (FAMElets) that can be implemented in any Turing-complete language and offer higher performance at equal or higher resource expenditure.

2.10.1 Innovative Claims

The PRINCIPALS architecture will provide a novel mechanism for carrying out defensive cybersecurity operations (DCO) at a scale, pace, and precision that has never been done before. It will be able to detect and track malicious activity across a 5G architecture, and it will provide the tools for effectively and rapidly countering such activity. PRINCIPALS will allow operators to write defensive (as well as network diagnostic) tools, called AMElets.

PRINCIPALS supports two types of AMElets, allowing for a rich and programmable network environment. To support this powerful extensibility model, a novel architecture will be prototyped and designed, with safety and security guarantees derived from a combination of :

- Carefully constructed semantics.
- A domain-specific language (DSL) for expressing transient network computation.
- Strict resource management.
- Cryptographic protections.
- Distributed authorization.

Coordinated action across a network can be implemented in PRINCIPALS following a variety of possible models, including through AMElet propagation across nodes, explicit communication between AMElets, and command-and-control communication with an operator console or other centralized management system.

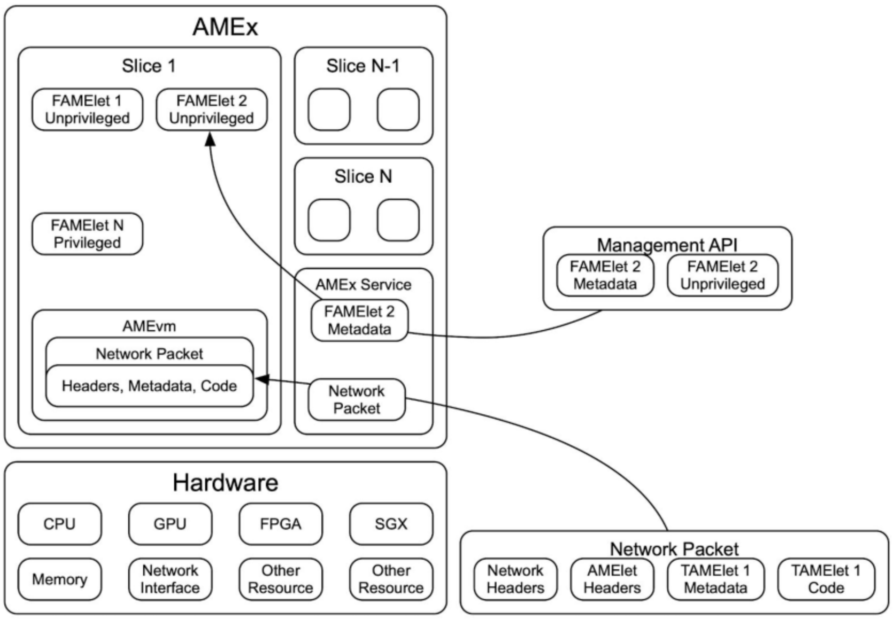
Another novel aspect of the PRINCIPALS effort is the use of multi-resolution emulation and simulation, combining Mininet [19] and the NS3 network simulation environment as part of its evaluation plan. Parameters like latency and overhead will be determined by the use of smaller-scale/higher-fidelity (SSHF) network emulations using actual PRINCIPALS runtimes. Following that, they are then used by larger-scale/lower-fidelity simulations that still use SSHF emulations in randomly chosen simulated network regions to scale to the desired evaluation scale program goals.

A final novel aspect of PRINCIPALS is its integration with end nodes (e.g., mobile handsets), leveraging secure execution elements such as Qualcomm's Secure Enclave to extend the ability to carry out DOC all the way to compliant end nodes. PRINCIPALS will be designed and prototyped in the form of software runtimes and services built atop open source operating systems (Linux and OpenBSD).

2.10.2 Threat Model and System Assumptions

While designing the PRINCIPALS architecture, it's necessary to make certain assumptions about the node on which the PRINCIPALS runtime operates and the capabilities of the attacker. It is assumed that the runtime itself is protected from malicious interference from enemies remote and local. By local we mean the attackers that are co-resident with a PRINCIPALS instance on a given 5G node. So PRINCIPALS is not concerned about hardware supply chain or close access attacks. In addition, it is assumed that the underlying platform offers protection against side channel attacks, that could reveal sensitive information that the runtime and AMElets use. Any algorithms used are outside the adversary's ability to successfully cryptanalyze and are implemented correctly. We, also, assume that the platform allows for a certain degree of extensibility. Finally we assume that there will be an attempt to use the PRINCIPALS infrastracture generally itself for malicious purposes. So it's only logical to assume that there will be at least some nodes that will initialize and attest to the integrity of a PRINCIPALS runtime.

2.10.3 Architecture



The PRINCIPALS node architecture, as illustrated in the figure above, is based on three pillars:

- **Flexibility**: PRINCIPALS provides complete but controllable programmability in every layer of the network stack. PRINCIPALS will offer the ability for defenders to inject their code from high-level, unprivileged applications, all the way to low-level, privileged system functionality, and potentially programmable hardware. This will provide the ability to defend against malicious attacks, analyze traffic and extract necessary information.
- **Performance**: Performing per-packet or per-flow computations in real-time (or near-real-time) requires efficient intra-node and inter-node communication, noteworthy processing capabilities for complex analytical computations, and sufficient memory resources. For PRINCIPALS to scale, we must minimize overheads, since executing multiple applications on the same infrastructure increases system complexity and the overall execution overhead.
- **Security**: PRINCIPALS is designed to support the injection of code in the data path, often by users that may not be fully trusted. Security models with great attention to detail are needed to increase usability as more precise policies can be specified and enforced. For example, the payload of packet will be accessible to some due to their credentials while not to some others. The specification as well as the enforcement of policy for the loaded

AMElets is where the challenge stands in this dimension. Security must be carefully balanced with the other two pillars of PRINCIPALS, performance and flexibility.

The main design choice for PRINCIPALS programmable architecture is how extensions are realized and the level of programmability it is capable of supporting. The perspective of PRINCIPALS is to support a wide spectrum of choices to maximize the benefits of the 5G infrastructure. These extensions (AMElets) require direct low-level access to packets or traffic flows, hardware features, and other resources, or are performance-sensitive, can be developed in any Turing-complete language and loaded into the network infrastructure, the PRINCIPALS nodes. Such *fat* AMElets (called FAMElets) can dynamically change a PRINCIPALS node's capabilities and operation. This covers functionality on one side of the spectrum and is very similar to the way programs or even operating systems can expand their functionality (e.g. plugins and extensions for Firefox or dynamically loadable modules for Linux) or the dynamic module model of active networks.

The second extensibility model of PRINCIPALS is the support of *thin* AMElets (called TAMElets), which consists of programs written in a domain-specific language (DSL). This limits the set of actions that TAMElets can take. TAMElets can be labeled as code-carrying packets that are executed as they traverse the network. They could possibly leave behind code that continues to execute until its allocated resources are used. TAMElets are an instance of the transient computation model of active networks. TAMElets rely on FAMElets and other PRINCIPALS core services to access traffic, computation, or other resources, through carefully constructed APIs.

PRINCIPALS can support traffic where all packets are code. This can be considered as an extreme IP network. To be more precise, if we think of IP packets as simple programs that invoke router functionality (e.g., store-and-forward, dropping of packets, etc), PRINCIPALS packets extend this thought to packets that contain actual code that is translated by the PRINCIPALS nodes. Depending on the services we wish to get from PRINCIPALS the degree of the extensibility and programmability will be determined. Resource usage, latency, should be taken into account. TAMElets can be thought as the dynamic parameterization/configuration and control of FAMElet-provided capabilities, in the context of a particular DCO mission. The design of these carefully constructed APIs and the delineation of the functionality between FAMElets and TAMElets in the initial prototype will be the subject of research and continuous refinement.

Below are described the key components of PRINCIPALS in more detail.

AMEx: The PRINCIPALS core component is the AME execution runtime . AMEx consists of a number of services and APIs and is responsible for controlling and mediating all accesses and calls between components (hardware or software). AMEx provides:

- The space where all FAMElets execute.
- An interpreter (AMEvm) for the execution of TAMElets written in AMEis (more details on AMEvm and AMEis are presented down below).
- Facilities for loading and unloading AMElets through the AMEx Service.
- Guarantees that AMElets do not interfere with each other.
- Intercession of communication between AMElets, as well as between AMElets and hardware resources of the 5G node (memory, network interfaces, CPU, accelerators and other computing devices like GPUs or FPGAs, trusted execution environments like SGX and TrustZone, etc.).
- Grouping of AMElets into slices
- Overall resource management per slice (or an interface to another resource management facility on the platform, if one exists).
- Guarantees that all operations are in line with the security policy that the infrastructure operators has set and the credentials the AMElets carry with them. More details about AMElet authorization will be given below.

AMEvm: is a simple virtual machine that can run the mobile code carried by the TAMElet. It is a simple interpreter that executes the instructions defined in the AMEis instruction set and serves as a sandbox for executing the code, relaying calls to FAMElet functionality that inhabits inside the AMEx. While for most purposes we expect running in simple interpreter mode, we do plan to investigate the potential use of Just-in-time (JIT) compilation of AMEis code to native (machine) code. Finally, it is important to note that there is a possibility of multiple instances of AMEvm running, as it is possible for multiple incoming packets to generate TAMElet code execution.

AMEis: is the instruction set that TAMElets are expressed in when transmitted in the network. The instruction set is very similar to the Berkeley Packet Filter (BPF) language, but it has two significant differences. Besides the standard instructions (arithmetic, logic, branch, and memory load/store), it is equipped with an interrupt instruction that allows it to imprison inside the AMEvm, and from there invoke functionality that has been defined in FAMElets. AMEis code can also register callbacks to itself, which are invoked by AMEx (via the AMEvm) when certain conditions are met. This allows an event-based on-the-fly programmability (and thereby orchestration) of services, including FAMElets, on a node. AMEis achieves two goals with this approach:

- It is able to alter the state of FAMElets, and altogether the state of a slice.
- It can rely on native and performant execution of heavyweight tasks while maintaining a lightweight and simple code profile.

Another difference from the implementation of the BPF language is that AMEis allows backward jumps. BPF implements only forward jumps for reasons of bounding the execution time. Forward jumps guarantee termination in time linear to the length of the program. We believe such a programming model is hard to use and very restrictive for the purposes of PRINCIPALS. We will implement upper bounds to the execution time of TAMElets using a different mechanism, by taking advantage of work done in prior projects of the Personnel in charge. Each TAMElet has a certain number of instruction cycles it can run and every instruction execution counts as one cycle. The AMEvm keeps track of the execution time and will terminate TAMElets that consume their allocated instruction cycles. Additional resource counters will be used to support wall-clock expiration, number of callback invocations, number of packets a TAMElet can generate, etc. Other TAMElets may increase the budget of already running (or resting) AMEis code, subject to proper authorization.

AMElets: come in two forms, as already mentioned. FAMElets are full-fledged code that is designed to execute on the PRINCIPALS framework, and specifically inside the AMEx. FAMElets are responsible for providing complex, performance-sensitive, and possibly privileged functionality. TAMElets can be thought of mobile programs that are designed to execute as they transit through the 5G network. They depend on FAMElets for much of their functionality and core services. TAMElets can be thought as scripting-like functionality that enables the dynamic configuration and composition of security (and other) services on a PRINCIPALS node.

TAMElets: are carried in network packets, with a special header that identifies them as a special kind of packet. The packet also encapsulates a payload of AMEis code along with metadata related to that code. The code payload is executed as it transits through the network. To be more precise, when reaching a PRINCIPALS node, the header is identified by the AMEx Service, which then proceeds to process the metadata. Metadata can include a variety of information like the TAMElet state, credentials that specify what permissions the TAMElet has, the slice wich the mobile TAMElet belongs to, etc. After the metadata is processed, the code of the TAMElets is given to an AMEvm for execution. While executing, TAMElets can alter the state of a slice via calls to FAMElets, or their own state (packet headers, metadata). TAMElet execution can end in a couple of ways. They may run out of cycles, in which case they terminate and AMEvm performs a cleanup, or they may explicitly exit or reach the end of their execution. When this happens, AMEx Service will check to see whether the current node is the final destination of the packet, in which case it will perform a cleanup. If not, the AMEx will forward the packet to the next destination. We anticipate TAMElets will be expressed in a programming language similar to PLAN.

TAMElet library: PRINCIPALS will provide a library of TAMElets for common network functionality. For example, update of state in common FAMElets, operations like collecting statistics from all slice nodes, modification in the functionality of FAMElets, heartbeat messages, PRINCIPALS node and AMEx control.

FAMElets: are software modules that execute inside AMEx. Their functionality may be different, depending on the needs of the services we need to deploy on PRINCIPALS. For example, a FAMElet can be a firewall module, a deep packet inspection system (e.g., Snort), a machine learning inference engine, a network monitoring system, or any other custom service. FAMElets can be written in any Turing-complete language.

Unprivileged FAMElets : In their most common form, FAMElets will be user-level programs that run inside a network slice. They can, and often will, interact with other FAMElets within the same slice, inside one or multiple PRINCIPALS nodes, to provide their services. Their access to slice resources will be checked by their credentials. For example, a FAMElet may not be allowed to access certain types of packets or traffic flowing through the slice.

Privileged FAMElets: When FAMElets require deeper system or network access, operations will be carried out by privileged FAMElets. These will be kernel-level modules running inside a slice. As such, they will typically have access to the whole resources of the slice, within and across multiple PRINCIPALS nodes. They will also be able to control the behavior of unprivileged FAMElets. They will have full access to the network resources of the slice, its computing capabilities, storage, etc. Typically, when a service has real-time needs, it may rely on privileged FAMElets, as they are closer to the hardware. Since privileged FAMElets are kernel-level modules, they are written in a typesafe language, and pointers and jumps are controlled. This is implemented in two different ways. Privileged FAMElets carry source code that is compiled by a trusted compiler within AMEx before it is loaded inside the PRINCIPALS operating system kernel. Alternatively, FAMElets carry object code that AMEx will patch to limit pointer access and jump targets. PRINCIPALS could allow for privileged FAMElets to carry code that will be executed at the GPU, or bitstreams that can be loaded onto the FPGA within a constrained area of the programmable fabric.

Management FAMElets : This type of FAMElet will be designed to run outside network slices, and inside the host operating system of the PRINCIPALS node. Operators will use these nodes of the 5G network infrastructure to changer or even extend the behavior and capabilities of the PRINCIPALS node. For example, they can update or patch the node and the AMEx. **Credentials**: con-

tain information about the resource constraints and permissions of the AMElets (TAMElets or FAMElets). Specifically, they inform the system how many cycles the AMElet is allowed to consume, how much memory it is permitted to allocate, what traffic it is allowed to monitor or modify, etc. The credentials are checked when the AMElet is loaded, to determine the type of AMElet. They are checked again in every API call performed by the AMElets. Credentials for FAMElets are typically created by managers of slices. Credentials for TAMElets are typically created by users that want to run their applications in a slice. The maximum rights in those credentials are capped by the type of user. We envision this to be a process that takes place out-of-band. As a starting point, we will use the KeyNote trust management system for authorization credentials in PRINCIPALS.

Resource management : This takes place in PRINCIPALS at a number of layers, across PRINCIPALS nodes and also within them. Within a node, resources are controlled within and across slices. Finally, resources are controlled at the level of the OS and hardware. The enforcement points depend on credentials to determine whether a resource (or how much of it) is available for use by an AMElet. As a whole, a 5G slice is given a resource budget (for network usage, memory usage, compute usage, etc.) to operate on all PRINCIPALS nodes. While this serves as an upper bound, the fact that the users of the slices can generate packets (TAMElets) that can traverse the network, perform computation, and generate new TAMElets, complicates resource management enforcement. To enforce a global security policy on resource consumption, we rely on the AMEx services of each PRINCIPALS node. When created, each TAMElet starts with a resource budget allocation. This allocation is encoded in the metadata it carries with it, in the form of a cryptographically signed credential. This makes it immutable to non-authorized entities. As the TAMElet jumps from node to node and executes, this resource budget is reduced. This is done by the AMEx Services, by modifying the credentials of the TAMElet to reflect the resource usage. In the event a TAMElets generates a new TAMElet, the resource budget is split between them (the exact split is subject to policy and the parent TAMElet). For example, assume a TAMElet is generated and is issued a credential with a policy that gives it 100 hops between PRINCIPALS nodes, 1000 compute cycles on AMEvms, and 200 calls to FAMElet services. At its first hop, the TAMElet uses 50 compute cycles and issues 2 FAMElet calls. As it propagates to the next node, it is issued a new credential stating that it still has 99 hops, 950 compute cycles, and 198 FAMElet calls. If at the next node it consumes another 50 cycles and 2 FAMElet calls, and then it generates a TAMElet, the two TAMElets will now share the resources, and the two newly issued certificates may each have 49 hops, 450 compute cycles, and 98 FAMElet calls. The above is an oversimplified example that does not take into account factors such as the cost of the FAMElet calls, but it serves to illustrate the concepts that we will be pursuing in PRINCIPALS.

Chapter 3

Primitives

Having a better understanding about how the 5G Network is implemented and how it relates to Kubernetes, we can now move on to the main focus of the thesis, the Network Security Primitives. In this chapter, it is presented the first part of this thesis' contribution. These primitives, designed and implemented, consist of a variety of defensive mechanisms that will provide PRINCIPALS the ability to neutralize any possible cyber threat on the 5G network. These primitives, or FAMElets as they are named inside the PRINCIPALS project, will be deployed inside the PRINCIPALS framework through carefully constructed APIs. Below we present the five network security pimitives that were designed and implemented as a part of this thesis.

3.1 Snort

`Snort` is an open source NIDS created by Martin Roesch in 1998. `Snort` is a free Network Intrusion Detection system software for Linux and Windows, that detects cyber threats like malicious packets, threats on Internet Protocol networks etc. It can perform real time traffic analysis and packet logging on IP networks. `Snort` relies on rules to determine what kind of malicious behaviour it should look for in a packet. These rules are stored in logging files that can be modified by a text editor. The rules are grouped in categories and depending on each category, they are stored in separate files. These files are then included in a main configuration file called `snort.conf`. `Snort` reads these rules at start-up time and builds internal data structures or chains to apply these rules to captured data. `Snort` comes with a rich set of pre-defined rules to detect intrusion activity and anyone can add their own rules or remove some of the built-in rules to avoid false alarms. The rules are proportional to the processing power that is required to process captured data in real time. `Snort` mainly consists of four components:

- Data sniffers

- Pre-processor
- Detection engine
- Alarm system

A packet read from the network card is first processed by the pre-processor. Then through rule detection packet in the detection engine. If the packet matches the rule, it will be processed in accordance with the rules.

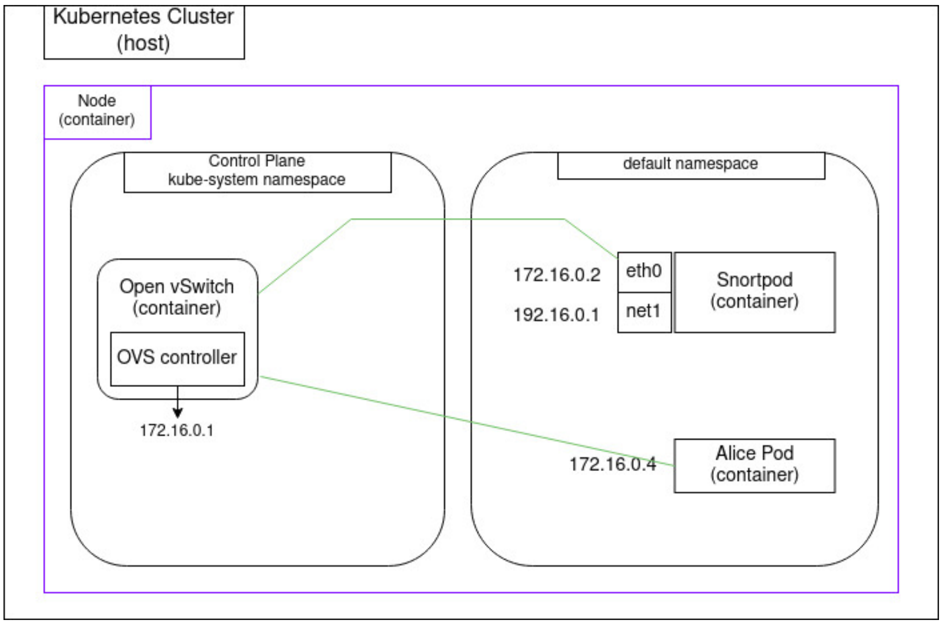
In this Primitive, `Snort` is being containerised, in other words a version of `Snort` is created inside a `Docker` container that has the same functionality. Given the rules `Snort` has been configured with, this primitive has the ability to monitor and find malicious traffic in the pod. In order to create a containerised `Snort` version, a `Docker` image must be created. Using a `Dockerfile` and taking an already existing image by using the FROM command inside, the Dockerised `Snort` starts its creation. The functionality of `Snort` is added by the installation of many necessary packages. Here is the whole `Dockerfile` :

```
FROM ubuntu:20.04


ENV DEBIAN_FRONTEND noninteractive
ENV NETWORK_INTERFACE eth0

RUN apt-get update && apt-get -y install \
    wget \
    build-essential \
    gcc \
    libpcre3-dev \
    zlib1g-dev \
    libluajit-5.1-dev \
    libpcap-dev \
    openssl \
    libssl-dev \
    libnghttp2-dev \
    libdumbnet-dev \
    bison \
    flex \
    libdnet \
    autoconf \
    libtool \
    nodejs \
    tcpdump \
    npm

WORKDIR /opt

ENV DAQ_VERSION 2.0.7
RUN wget https://www.snort.org/downloads/snort/daq-${DAQ_VERSION}.tar.gz \
    && tar xvfz daq-${DAQ_VERSION}.tar.gz \
    && cd daq-${DAQ_VERSION} \
    && ./configure; make; make install

ENV SNORT_VERSION 2.9.18.1
RUN wget https://www.snort.org/downloads/archive/snort/snort-${SNORT_VERSION}.tar.gz \
    && tar xvfz snort-${SNORT_VERSION}.tar.gz \
    && ls \
    && cd snort-${SNORT_VERSION} \
    && ./configure; make; make install

RUN ldconfig

ADD mysnortrules /opt
RUN mkdir -p /var/log/snort && \
    mkdir -p /usr/local/lib/snort_dynamicrules && \
    mkdir -p /etc/snort && \
    cp -r /opt/rules /etc/snort/rules && \
    mkdir -p /etc/snort/preproc_rules && \
    mkdir -p /etc/snort/so_rules && \
    cp -r /opt/etc /etc/snort/etc && \
    touch /etc/snort/rules/white_list.rules /etc/snort/rules/black_list.rules

RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
    /opt/snort-${SNORT_VERSION}.tar.gz /opt/daq-${DAQ_VERSION}.tar.gz

COPY myjsscript.js /var/log/snort/myjsscript.js
RUN chmod +x /var/log/snort/myjsscript.js
```

Finally with the docker build command, all of this necessary packages is being applied inside the container and when the building is done, the container is ready for use. By using the Docker containers, Snort can run individually inside the Kubernetes infrastructure as a pod (Snortpod), with the application of a simple yaml file. For this to happen the image of the Snort container that was created must be pushed either in Docker Hub or our local repository. As an extend to this, with the help of the Antrea Agent, the whole traffic of the node is being mirrored to the Snortpod. This is achieved with the use of Init container, a useful component of the Kubernetes environment that makes it possible to run a specific task before the initialization of any pod or deployment.

The configuration of the image required for the Init container is similar to the previous one. The image that we use in our Dockerfile as a base image, is an Antrea image though. That's because this deployment works by having installed the Antrea project inside and by using the Volume Mount components with the Antrea agent of K8s. Inside this shared file, we store a bash script that essential makes the Antrea Agent to apply the mirroring rule to the bridge when it executes. The Init container's task is the execution of this script. In that way the mirroring can be implemented. This is whole configuration is implemented in the yaml file of the Kubernetes environment, given the fact that we have already created a containerised Snort deployment and a containerised mirroring deployment. It's worth mentioning, that when the Antrea agent applies the rule, the mirroring is realized in the first interface of the SnortPod, the eth0 interface. The SnortPod then, can no longer use this interface for nothing else than mirroring. So if there is a need to communicate with an external or an internal user, the Snortpod will have to use its secondary interface. The secondary interface is created with the use of Multus inside the yaml file. The necessary Dockerfile and yaml file alongside with a rule file can be found here. Inside also exists the code and yaml file required for the mirroring. A very detailed explanation is also provided in the README file.

3.1.1 Example



We assume that there is a pod that creates traffic inside the Cluster, called Alice pod. Whatever traffic the Alice pod creates it's being mirrored to the SnortPod. The Snortpod primitive has logged all the predefined information about the traffic that wants to monitor. The rule that is configured with is this:

```
alert ICMP 172.16.0.4 any -> any any ( msg:"Sample alert"; sid:1000001; rev:1; )
```

The meaning of this rule is that any time this IP(172.16.0.4) creates ICMP traffic, it will be logged. This is the IP of the Alice pod though. That means that if the Alice pod creates any ICMP traffic, this traffic must be logged. The Alice pod starts pinging at google.com. The Snortpod monitors this traffic and logs it in the `alert` file inside the container's `/var/log/snort` directory. The figure above shows a representation of this scenario in a `Kubernetes` Cluster.

3.2 DNS Sinkholing

DNS Sinkholing is a mechanism used to protect users that attempt to connect to known malicious or unwanted domains by intercepting their DNS request. Then it returns a false, or even better a fixed IP address. This fixed IP address will point to whatever the system administrator defines it. This technique can be used to prevent users from connecting or communicating with known malicious destinations (botnets, etc) in a secure workplace environment.

The input to the DNS Sinkholing primitive is a list of pairs (domain,IP address), where *domain* is the domain to be sinkholed and *IP address* is the address the primitive will inject at the request. The implementation of the primitive takes advantage of the CoreDns plugins. By using the rewrite and hosts plugin, the job of this primitive becomes very easy. When the pod, named DNS-Support, is deployed, it runs a series of scripts that do the following :

- Checks the input file.
- Reads the input file and finds out the domains that need to be sinkholed.
- Having the domains needed, it then writes a new Corefile for the CoreDns and applies it.
- Then it restarts the CoreDns with the new Corefile. The new Corefile has the requested domains written in the Hosts plugin, next to the IP we want to sinkhole them to. It also covers the top-level domains case with the use of the rewrite plugin.
- Finally, it periodically checks the input file for changes. In the event of a change, the Primitive will restart the whole drill with the new input file.

For this implementation to take place, the creation of the DNS-Support pod has some background procedure. For starters, the functionality of the pod must be Dockerised. This happens again, by creating a `Dockerfile` that takes an Ubuntu image and it's added some packages to be installed, like Node.js (which is the language that is implemented in), npm, etc. The scripts that this pod will run are also added in the `Dockerfile`. These scripts are three in total and provide the necessary functionality the DNS Sinkholing Primitive needs.

- **rewriteCorefile.js** is the script that creates a new Corefile with the host plugin filled with the domains that exist in the input file.
- **start.js** is the script that is immediately executed when the pod is deployed. This script uses the rewriteCorefile.js script to create the new Corefile that will be used in the new CoreDNS and then uses it to restart the CoreDNS with it.
- **run.js** is the last script that runs for the rest of the lifecycle of the DNS Support pod and monitors the input file with the domains. If any changes are made in that file, the run.js script will basically run a copy of the start.js script that will redo everything from the start.

This Pod has to have the ability to make changes to the Cluster, in other words to be able to execute kubectl commands. This happens by giving it the appropriate packages installed in the `Dockerfile`. Here is the whole `Dockerfile`:

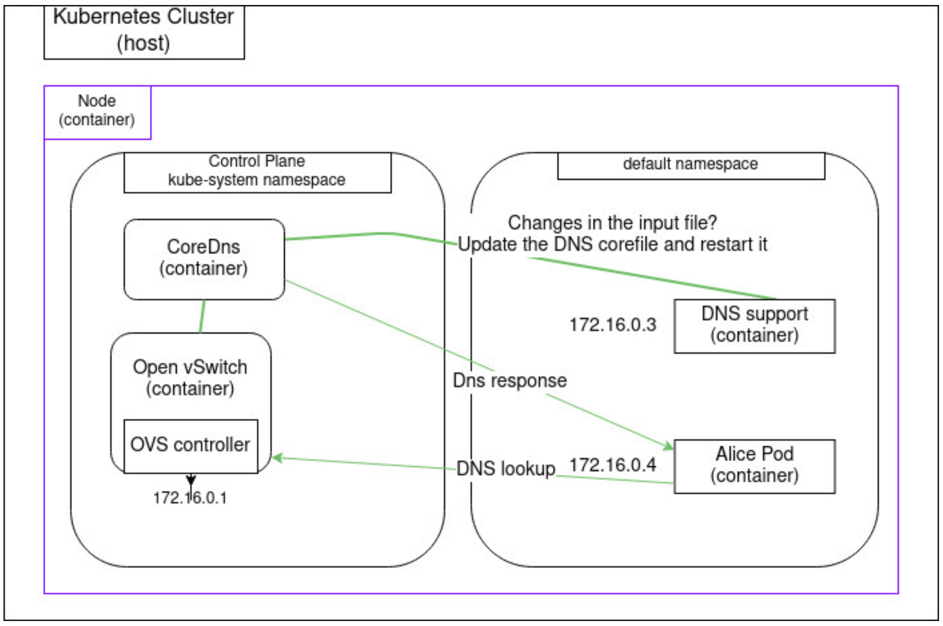
```
FROM ubuntu
ARG DEBIAN_FRONTEND=noninteractive

RUN  apt-get clean
RUN  apt-get -y update
RUN  apt-get -y upgrade
COPY script.js /home/script.js
RUN chmod +x /home/script.js
COPY run.js /home/run.js
RUN chmod +x /home/run.js
COPY start.js /home/start.js
RUN chmod +x /home/start.js
RUN apt-get install -y nano
RUN apt-get install -y npm
RUN  apt-get -y install curl
RUN  curl -LO "https://dl.k8s.io/release/$(curl -L
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl"
RUN  curl -LO "https://dl.k8s.io/$(curl -L -s
https://dl.k8s.io/release/stable.txt)/bin/linux/amd64/kubectl.sha256"
RUN  install -o root -g root -m 0755 kubectl /usr/local/bin/kubectl
RUN  apt-get install -y nodejs
CMD ["sleep", "infinity"]
```

Now the Dns-Support is ready to be built and used as a pod inside the `Kubernetes` infrastructure. By having the image of it in our local repository, a yaml file is needed to deploy the pod in our Cluster. The yaml file is deployed in the Cluster via the command:

```
kubectl apply -f
```

With the pod's initialization, the CoreDns should restart as well. The necessary Dockerfile and yaml file alongside the code needed to implement the DNS Sinkholing primitive, can be found here. A very detailed explanation is also provided in the README file.

3.2.1 Example



We assume again that there is a pod that creates traffic inside the Cluster, called Alice pod. At the same time the DNS support Pod exist in the node, with a list of domains to be sinkholed. Suddenly the Alice pod starts pinging at google.com. This domain is not listed inside the input list of domains to be sinkholed. So the CoreDNS, who doesn't have google.com inside its Corefile's host plugin, returns google's legit IP, 142.250.179.142. So now the Alice pod is pinging at 142.250.179.142. The system's administrator then decides that he/she wants to ban google from the employees. So google is added in the input file with an IP of 127.0.0.1. At that time, the DNS Support pod finds out that there has been a change in the input file. Immediately, it creates a new Corefile using the new input. When this job is done, it applies it to the CoreDNS and it restarts it. Now the CoreDNS has a new list of domains to be sinkholed and google.com is one of them. The Alice pod suddenly stops pinging at 142.250.179.142 and starts pinging at 127.0.0.1.

The figure above shows an representation of this scenario in a `Kubernetes` Cluster.

3.3 Flow Recording

Just like the `Snort` Primitive case, in this Primitive we want to create a mirroring of the traffic to our pod. This means that besides the containerization of the Analyzer pod, the pod that is going to provide the functionality in this primitive, there is a need for a mirroring container. This will be done again with the use of the Init containers component of Kubernetes. The procedure is similar. From a Dockerfile, we are going to create a new image of Antrea, that will have the ability to share a Volume with the Antrea Agent of our Kubernetes Cluster. Then a simple bash script is added, that will provide the functionality needed to create the mirroring. Having the initialization process completed, the Analyzer pod now has the whole traffic of the node mirrored to itself. The functionality that we need from the Analyzer pod now is the following:

- Reads the Input file from the API and extracts the domain names we want to monitor.
- Monitors DNS traffic from a given OVS port and extracts DNS requests.
- If there is a match with a domain name from the input file, the analyzer will create an object with specific information about the packet.
- Then it will forward this object to the logger pod, a pod created in order to store the information.

First, we started with the known procedure. A new image is created, with the help of a Dockerfile and an image of Centos. Then the requirements are added in the Dockerfile along with the Python script that is designed for this exact job. Here is the whole Dockerfile :

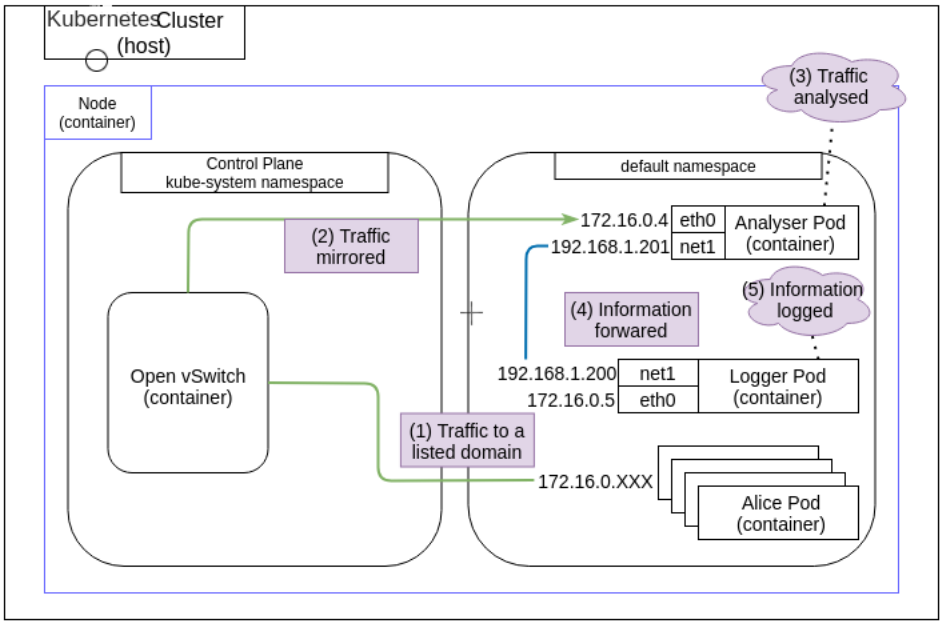
```
FROM centos:8

COPY domains.txt monitor.py requirements.txt /tmp/

RUN yum -y install python3-pip python3-devel
RUN yum -y install gcc gcc-c++
RUN pip3 install -r /tmp/requirements.txt
```

With the help of Scapy, a sniffing process begins with a filter of *port 53*, since we're only interested in DNS Requests. When a DNS request matches with a domain from the list of domains we get from the input file, the script creates the designated object with the information. When this job is done, through a TCP socket, the pod will communicate with the Flow-Server pod, or any other logger pod that we desire, and will exchange these information. The IP and the port of the logger pod to communicate with, are given to the python script as an input as well.

As mentioned before, it's important to keep in mind that when a pod has one interface that has traffic mirrored to it, it can no longer use this interface to communicate. As a result to that, the Analyzer pod will have to create and use a secondary network interface to communicate with the logger pod. The logger pod, on the other hand, will also have to have a secondary interface, because the communication between a primary interface and a secondary interface is impossible.

The secondary interface is created with the use of `Multus` inside the yaml file. The necessary Dockerfile and yaml file alongside the code needed to implement the Flow Recording primitive, can be found here. A very detailed explanation is also provided in the README file.

3.3.1 Example



We assume there is a pod that creates traffic inside the Cluster, called Alice pod. At the same time the Analyzer Pod exists in the node, with a list of domains to be monitored. Suddenly the Alice pod starts pinging at google.com. This domain is not listed inside the input list of domains to be monitored. So the Analyzer Pod, who doesn't have google.com inside the list of domains to analyze, does nothing. All of a sudden, the Alice pod is pinging at example.com. This domain, example.com, exists inside the input list of domains of the Analyzer pod. The Analyzer pod has all the traffic of the node mirrored to its eth0 interface. In this way, it monitors all the DNS Lookups of the node. So when the Alice pod pings at example.com, the Analyzer monitors that DNS lookup and because example.com is a domain that it is interested in, it creates an object. This object consists of useful information about the DNS request, like the time it was made, the IP and MAC addresses of the source and destination of the packet, etc. Right after, it sends it to the logger pod, through the secondary interface, net1. The logger pod stores this information in a logging file for the system administrator to access. The figure above shows an representation of this scenario in a `Kubernetes` Cluster.

3.4 DDoS Detector

A Distributed Denial of Service (DDoS) is a kind of attack that plans to overflood the traffic of server, service or network. The way to achieve such a malicious act is by overwhelming the target or its surrounding infrastructure with a flood of Internet traffic. You can see this kind of attack, like an unexpected traffic jam that just blocks the way of normal passengers and delays them from their destination, if they even reach it eventually.

DDoS attacks are carried out with networks of Internet-connected machines. These networks, that mostly consist of computers or other electronic devices, are usually infected by some kind of malware that enables the attacker to control them remotely. Each one of these devices is called a bot in the department of cybersecurity, and the whole group of them is called a botnet. When the attacker has established connection with every bot in his botnet, he is ready for the attack. By sending remote instructions to the botnet, the server, that he has planned his attack to, starts flooding in traffic. The server that has just received way too much traffic, becomes overwhelmed and starts to show a denial-of-service to normal traffic.

For the purpose of defending such an attack, this Primitive was created. DDoS Detector is a primitive that uses a Golang implemented script to defend against a DDoS attack. So this how this primitive works. Starting as you can probably guess, we are going to create a mirroring to the first interface of the Pod. The procedure is the same with previous primitives. A mirroring container is created from the image of `Antrea`. Via the Init containers component the `Kubernetes` provides along with a volume-mount with the `Antrea` agent, a mirroring of the traffic is created to the first interface of the pod (the eth0 interface). Having the initialization ended, the pod is ready to start. The pod works by executing a Golang implemented script that takes as input the following :

- The interface to read packets from.
- The filename to read from, overrides the interface option if it exists.
- The snap length (number of bytes max to read per packet, default value is 65536).
- The packet threshold (its value is packets per second, default value is 150).
- A Boolean option to check if it is an syn attack(default value is false).
- The path to the log file.
- The flow-server connection in format IP:Port.
- The IP and the port of the secondary network that listens for connections in format IP:Port.

- The command to execute when a malicious behaviour is detected e.g. block, tarpit, etc.
- Arguments to pass to the command you want to execute.

Golang is a multi-threaded language, so many processes are taking place simultaneously. Using the pcap library, we create a monitoring interface that starts with a BPF filter of net 127.0.0.1, in other words that monitors the loopback. At the same time a TCP socket is open for listening incoming traffic. This TCP socket and any other incoming or outgoing communication is happening through the secondary interface. The secondary interface is created with the use of `Multus` inside the yaml file. Whenever the Canary, a pod that already exists in the PRINCIPALS project and won't be explained thoroughly, detects any *weird* traffic, it will send a message to our pod, through the TCP socket, with the IP that creates it.

In every message our listener gets, each IP is extracted and added to the array of IPs that are being monitored. Bear in mind that all of this is happening with the necessary use of Mutexes(locks and unlocks) since Go is a multi-threaded language, as already mentioned. In every new IP we get, the BPF filter changes to the appropriate value in contemplation of monitoring this IP as well. Simultaneously, two timers have been set, to check periodically the incoming traffic of the IPs that we monitor. Through a variety of *ifs*, each packet is adding value to a number of counters (tcpCounter, udpCounter, HttpCounter, etc), depending on its characteristics. At the end of the first timer, we create some variables. Each counter is multiplied by 100 and is divided by the total number of packets we got in this time. Then a comparison of each new variable against a fixed number is made. If a variable is greater than a fixed number, then we known we are under a certain type of attack. In that case, our pod will have to inform the Flow-Server, an important pod of the PRINCIPALS project that has the ability to *talk* to the bridge, about the attack that we are under. So the pod will send to the Flow-server, again through a TCP socket, the appropriate information (IP of the malicious pod, command to be executed by the bridge like block, tarpit etc), and the Flow-Server with its turn will apply the command to the bridge.

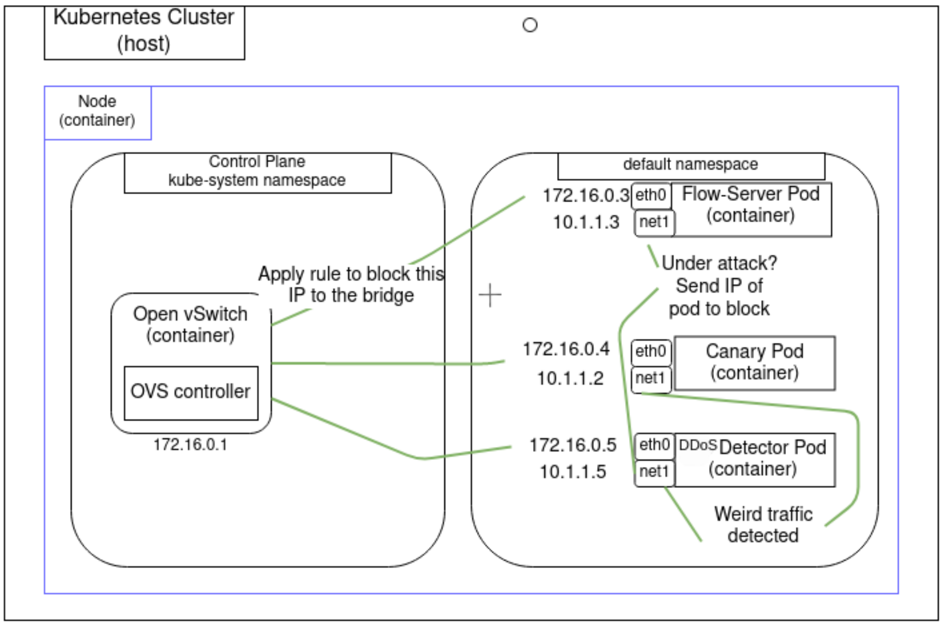
But that's in case an attack is happening. If none of the conditions is fulfilled, then it's going to keep on repeating this process until the second timer strikes. When the second timer strikes, the Timeout timer, then it's decided that the IP that is being monitored is not malicious and the array of monitored IPs, clears it. This results in the change of the BPF filter to net 127.0.0.1 or to the rest of the IPs existing inside monitored IPs array, until traffic from the Canary comes.

All of this, are included in the Golang implemented script our pod will execute. This means that a container should be created with all the necessary packages and dependencies, alongside this script. This is implemented inside the following `Dockerfile`:

```
FROM projects.registry.vmware.com/antrea/antrea-ubuntu:v1.0.0

RUN apt-get update && apt-get install -y \
    iperf3 \
    libpcap-dev \
    sudo \
    nodejs \
 && rm -rf /var/lib/apt/lists/*

COPY bin/ /home/tsi/bin
COPY scripts/ /home/tsi/scripts
```

With the help of a Makefile that creates this repositories, as well as the executables inside, the container is ready to be applied inside the cluster. The necessary Dockerfile and yaml file alongside the code needed to implement the DDoS Detector primitive, can be found here. A very detailed explanation is also provided in the README file.

3.4.1 Example


Kubernetes Cluster (host)
Node (container)
Control Plane kube-system namespace
Open vSwitch (container)
OVS controller
172.16.0.1
Apply rule to block this IP to the bridge
default namespace
172.16.0.3 eth0 Flow-Server Pod (container)
10.1.1.3 net1
Under attack? Send IP of pod to block
172.16.0.4 eth0 Canary Pod (container)
10.1.1.2 net1
172.16.0.5 eth0 DDoS Detector Pod (container)
10.1.1.5 net1
Weird traffic detected

We assume that there is a Kubernetes cluster with the above configuration. So there are at least three pods deployed inside the default namespace. These three pods are the Canary pod, the Flow-server pod and the DDoS detector pod. There is nothing weird in the traffic all the cluster and everything is running smoothly. All of a sudden the DDoS detector receives a message in its secondary interface. Its eth0 interface is only for mirroring. The Canary pod has detected some weird traffic that needs to be examined. The DDoS detector pod, extracts an IP to monitor from the message it received. It starts tracking the traffic that is created by this IP. Inside two seconds of monitoring this IP's traffic, the DDoS detector finds something out. This IP has created an extraordinary amount of SYN traffic. There's no time to lose. The cluster is under a SYN attack. The DDos Detector pod creates a message for the Flow-server and sends through its secondary interface, net1, to the secondary interface of the Flow-server. The messages consists of the IP address of the malicious pod that has created this traffic, alongside a rule. The rule is to block it. The Flow-server receives the message and immediately applies the rule to the bridge. The IP is blocked and the cluster is safe.

The figure above shows an representation of this scenario in a `Kubernetes` Cluster.

3.5 DGA Detector

DGA stands for Domain Generation Algorithm and is a technique used by attackers to generate new domains (names and IP addresses) for malware's C&C servers. Executed in a way that seems random, it makes it nearly impossible for cybersecurity operators to detect and defend against the attack. Changing domain names helps the attackers by keeping their servers from being blocked from their targeted victims. The main idea is to have a program that produces random domain names that the malware can use and quickly switch between. Security operators usually block or take down the malicious domains that malware creates, so switching domains enables them to continue the attack.

In order to defend against this kind of attack, the use of a machine-learning module is a must in this Primitive. Starting from the top, we need to create a mirroring of the traffic of the whole node, so that we can extract every DNS request. Without further ado, the procedure has been explained thoroughly before, we create a mirroring container and initialise our pod, the DGA Detector pod, with it. Now that we have the whole traffic mirrored we are going to need a script that monitors that traffic.

Our container starts with a basic Centos image and then it's added some necessary requirements. First, we add a dga.model file to our container, a `Keras` model that is created for the exact purpose of finding domains that have been created by an algorithm. Now that we have a machine-learning model to use, we need to create an script for the pod to use. The implementation of the container is concluded with this script and it's ready to build. The `Dockerfile` the DGA Detector primitive uses is the following:

```
FROM centos:8

COPY dga.model monitor.py requirements.txt /tmp/

```

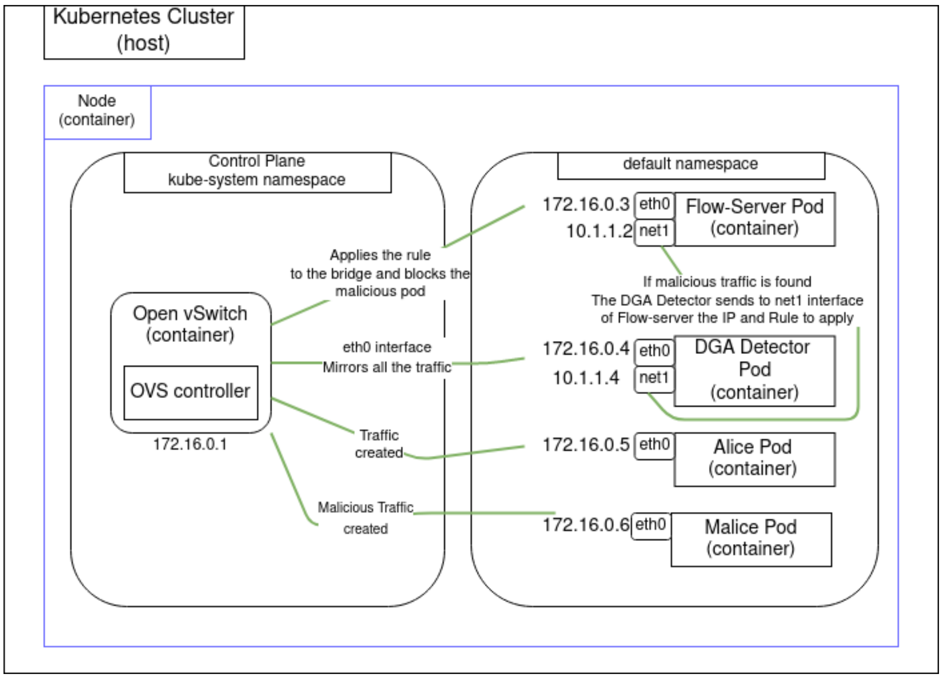
```
RUN yum -y install python3-pip python3-devel
RUN yum -y install gcc gcc-c++
RUN pip3 install -r /tmp/requirements.txt
```

A Python script is implemented, the monitor.py as you can see in the `Dockerfile`. This script uses the Scapy library in order to create a monitoring, with a filter of *port 53*, since the DNS request are the only thing we're interested in. Beside the sniffing that Scapy will provide, a class named DGA Detector is created, that takes a domain name and *breaks* it to a list of ints. Then, by using the appropriate padding, this list of ints is passed to the `Keras` model. The `Keras` module will return a prediction, a number between 0 and 1, that will be the possibility of this domain to be a domain generated by an algorithm. If the prediction is higher than 0.5, we choose to deal with the domain as a domain generated by an algorithm.

Having the ability now to tell whether a domain is legit or not and having the whole traffic mirrored to our first interface, we can deal with a DGA attack. For every DNS request that is made in our node, our pod will get the domain name and the IP of the requested domain and will check if it's valid or not. If it's not, then through a TCP socket, our pod will communicate with the Flow-Server and it will send him the IP of the domain alongside a command with the value *block*. The Flow-Server is going to apply the command with the specific IP to the bridge, and this domain is going to be blocked from our Cluster.

It's important not to forget, that since our first interface is used for the mirroring, the communication between our pod and the Flow-Server pod, is carried out through the secondary interface of those two pods. The secondary interface is created with the use of `Multus` inside the yaml file. The necessary Dockerfile and yaml file alongside the code needed to implement the DGA Detector primitive, can be found here. A very detailed explanation is also provided in the README file.

3.5.1 Example


Kubernetes Cluster (host)
Node (container)
Control Plane kube-system namespace
default namespace
Open vSwitch (container)
OVS controller
172.16.0.1
Applies the rule to the bridge and blocks the malicious pod
eth0 interface Mirrors all the traffic
Traffic created
Malicious Traffic created
172.16.0.3
10.1.1.2
eth0
net1
Flow-Server Pod (container)
If malicious traffic is found The DGA Detector sends to net1 interface of Flow-server the IP and Rule to apply
172.16.0.4
10.1.1.4
DGA Detector Pod (container)
172.16.0.5
Alice Pod (container)
172.16.0.6
Malice Pod (container)

We assume there are two pods that create traffic inside the Cluster. The one is called Alice pod and creates normal traffic. The other one is called Malicious and creates malicious traffic. To be more precise, it's making an attack by using Domain Generation Algorithm. At the same time the DGA Detector Pod exists in the node, as well as the Flow-Server pod. The DGA Detector monitors the whole traffic of the node, by having all the traffic mirrored to its eth0 interface. The DGA Detector monitors the DNS lookups and decides whether or not a domain is created by an algorithm. When the DNS lookup of the malicious pod reaches the DGA Detector pod, the Detector checks the validity of the domain. It predicts that this domain is generated by an algorithm. Right away, it sends a message to the Flow-server's secondary interface, through its net1 interface. This message consists of the IP of this domain and a command. It's a blocking command. The Flow-Server receives the command and immediately applies it to the bridge. The bridge blocks the received IP address. The attacker has no longer the ability to slow the traffic.

The figure above shows an representation of this scenario in a `Kubernetes` Cluster.

Chapter 4

Language

In this chapter of the thesis, we are going to present the second and last part of this thesis's contribution. In order to create the necessary permissions needed for the implementation of TAMElets, we had to implement some language security techniques. Before presenting the tools used for the language security and the work done in order to use them, an introduction is given on why a language security technique like code analysis is so important.

In the department of security, code analysis is always a matter of great importance. The last few years, with the use of third-party libraries, it has been noticed an excessive number of attacks against the supply-chain [47], [46],[28]. Small problems, internal to the libraries, or even malicious code that can be introduced, create unprecedented obstacles. The majority of modern programs use thousands libraries with thousands of lines of code in total. Libraries, small in number of lines code, are used by hundreds of applications as well. As a result, such attacks are difficult to counter and affect a wide range of the applications ecosystem [25], [53].

A relatively recent example of this kind of attack at the event stream [3], [22], where the maintainer of a very popular package, inserted inside the package new code. This code's aim was to steal the security passwords of the wallet of a very popular cryptocurrency (Bitcoin) [6]. Code analysis is necessary in order to avoid such risks.

Right below is now presented the already existing tools used in this thesis to provide some security guarantees on language level of the framework. The two language tools are composed from a dynamic (§4.0.2) and a static analysis tool (§4.0.1).

4.0.1 Static Analysis

Third-party libraries make the development of large-scale software systems much easier, that is why modern software development relies heavily on them. Such reliance has led to an explosion of attacks [42], [46], [47], [50]. However, they often execute with crucially more privileges than needed to complete their task. Even if these libraries are not actively malicious, this additional privileges are often exploited at runtime via dynamic compromise. This exploitation can compromise the application or even worse the broader system on which the application is executing. Static analysis, also called static code analysis, is a method of computer program debugging. Static analysis is realized by examining the code without executing the program. The process provides an understanding of the code structure and can help ensure that the code adheres to industry standards. To address such problem a tool of static analysis is a necessity.

The tool that is going to be used to deal with this problem is MIR [52]. MIR introduces a fine-grained read-write-execute (**RWX**) permission model at the boundaries of libraries. Every field of an imported library is governed by a set of permissions, which developers can express when importing libraries. To enforce these permissions during program execution, MIR transforms libraries and their context to add runtime checks. As permissions can overwhelm developers, MIR's permission inference generates default permissions by analyzing how libraries are used by them. Applied to 50 popular libraries, MIR's prototype for JavaScript demonstrates that the RWX permission model combines simplicity and power:

- It is simple enough to automatically infer 99.33% of required permissions.
- It is expressive enough to defend against 16 real threats.
- It is efficient enough to be usable in practice (1.93% overhead).
- It enables a novel quantification of privilege reduction.

MIR manages to prevent real attacks without breaking library functionality while requiring minimal user effort and imposing minimal overhead on execution and compilation time. MIR is a lightweight addition to any contemporary developer's toolkit, complementing defense mechanisms.

4.0.2 Dynamic Analysis

In this section will be shown the tool used for the dynamic code analysis in the code implemented in the Network Security Primitives of this thesis. Lya [51] is a tool that provides Library-oriented Dynamic Program Analysis for JavaScript. Dynamic program analysis is a technique for obtaining information about a program and its execution. Lya provides a new type of dynamic analysis that targets modern dynamic languages such as Python, Lua and Javascript. This kind of dynamic analysis uses the capabilities of the module-import mechanism, and more specifically the fact that libraries are imported as text. At its core, Lya performs disassembly, transformation and re-composition of the imported libraries while maintaining their original functionality. During the transformation phase, it injects user-generated code, specific to each analysis, into the source code of the library. This user-generated imported code, combined with various context wraps and the use of proxies, makes it possible to detect interactions at the library level, without altering the runtime environment. This approach is implemented in Lya, a coarse-grained dynamic analysis framework that bolts onto a conventional production runtime as a library, written in Javascript. Lya offers 2–3 orders of magnitude faster analyses compared to conventional dynamic analysis systems. Analysis can be enabled to run during production, detecting problems and behaviors unique to these conditions.

4.0.3 Using Static and Dynamic Analysis Tools on PRINCIPALS

In order to use this tools, Lya and MIR, we had to make some changes to the Network Security Primitives. This tools are designed to work on Javascript implemented code. That means that each primitive had to be rewritten in Javascript in order to get analyzed. So all the scripts in all of the primitives(Analyzer, Sinkhole, DGA Detector, etc) had to be changed from the various language implemented (Golang, Python, etc) to Javascript, or to be more precise to Node.js, a runtime environment that's used to run JavaScript outside the browser. That being said, the whole process of each primitive had to be done again from scratch.

Each Network Security Primitive had to be reconstructed. Every container built, besides the ones already implemented in Javascript (DNS Sinkholing Primitive and the mirroring container of the Snort Primitive), had to rebuild in Javascript. Starting from the very start, the way to begin is by creating new containers. These new containers have more or less the same configuration. The only differences are the Node.js install, alongside with NPM [21]. NPM is a package manager for the JavaScript programming language. Every package needed for the script each Primitive used, or better saying, each container since mirroring was also implemented in Node.js, have to be installed in the Dockerfile of each container. Before the installation of the packages, the following command is executed first.

```
RUN npm init --yes
```

A json file named `package.json`, that contains all of the packages used in the script is added as well. The code of each primitive implemented in Node.js alongside the necessary yaml files can be found in here. A detailed explanation is also provided inside the according README files.

The Primitives are now ready to run in the same way the "old" ones worked. But in this section we will do the static and the dynamic analysis. What that means is that we need to install the tools that will be used on each Primitive's Dockerfile. Bear in mind, that most of the Primitives use mirroring, so they can't use external networking communication. So it is necessary to install the tools in advance. The installation of these tools is easily done by these two commands:

```
npm i -g @andromeda/mir-sa
npm i -g @andromeda/mir-da
```

The first one is for the static and the second one for the dynamic analysis.

The rest of the procedure for each Primitive to be implemented is the same, so no need to further analyse that. With the "new" Primitives now though, we can use the tools discussed above. Starting with MIR, we make a static analysis for each script. In order to have a deeper understanding, right below is the json file, named `static.json`, produced by the static analysis by MIR to the script that the Flow Recording Primitive uses:

```
{
  "/home/analyzer.js": {
    "Date": "r",
    "Date.now": "rx",
    "JSON": "r",
    "JSON.parse": "rx",
    "JSON.stringify": "rx",
    "console": "r",
    "console.log": "rx",
    "parseInt": "rx",
    "require": "rx",
    "require('argparse')": "ir",
    "require('argparse').ArgumentParser": "rx",
    "require('dns-packet')": "ir",
    "require('dns-packet').decode": "rx",
    "require('fs')": "ir",
    "require('fs').readFileSync": "rx",
    "require('net')": "ir",
    "require('net').Socket": "rx",
    "require('pcap')": "ir",
    "require('pcap').createSession": "rx",
    "require('pcap').decode": "r",
    "require('pcap').decode.packet": "rx"
  }
}
```

By running inside the Analyzer pod the following command :

```
mir-sa /home/analyzer.js &> /home/static.json
```

the `static.json` file is created. As you can see, inside the `static.json` file exists every library and every function of the library used inside the Analyzer script (`analyzer.js`). Right next to them, there is a string that contains the access permissions each element should be given. The possible permissions are:

- R, read.
- W, write.
- X, execute.
- I, import.
- any combination from the above.

Now that we know what permissions each library should have, we are going to use the file that includes them, the `static.json` file, to do a dynamic analysis using this file. In other words we are going to execute the script with the Lya tool, and give this file as an input. The command we use is :

```
mir-da analyzer.js --module-include '/home/analyzer.js' -e /home/static.json
```

If there are no security issues, then the script is going to execute without any problem. If more credentials are required for the execution of the script than the ones already defined inside the json file, then the execution is going to terminate.

That is how these tools are going to be used in PRINCIPALS. To be more specific, TAMElets are going to use them for their execution. TAMElets, which are special packets carrying code to be executed inside the PRINCIPALS node, will use this tools in order to know exactly what kind of credentials they need. TAMElets have the ability to reconfigure the whole node, so we must be very careful with the credentials we are giving to them. The tasks they are assigned with, always needs limited amount of permissions. If more than the ones MIR generates are asked, then the dynamic analyses tool, LYA, is going to terminate its execution. No security issues will be realized.

By generating the permissions that FAMElets need to execute we can set an upper-bound. No TAMElet will ever need permissions that FAMElets don't have. So when a TAMElet executes through LYA, by having the FAMElets permissions as an input, LYA will check if the TAMElets permissions are the same or a subset of them. If not the TAMElets' execution will be terminated.

Chapter 5

Evaluation

In this chapter we present the last part of this thesis' contribution, the evaluation the DDoS Detector primitive. This test objective focuses on testing the PRINCIPALS approach in defending the 5G network against DDoS attacks. The testing focuses particularly on the different variants of the Mirai botnet [35] attacks. A number of variants of the Mirai attacks and their combinations will be considered in the testing.

The fundamental question to ask is, *are defenses capable of detecting and removing a DDoS attack while ensuring that legitimate service continues at a user acceptable level during the attack?*. To answer this question, we need metrics that can effectively measure the performance of the defenses. The following metrics have been identified for testing the DDoS defenses:

1. **Percentage of attack traffic dropped**: How much of the attack traffic the DDoS defenses are able to detect and drop?
2. **Legitimate traffic's goodput**: How much of the good traffic is reaching the destination?
3. **Delay and loss rate**: What is the delay and loss rate of the legitimate packets?
4. **Percentage of legitimate packets delivered**: How high of packet delivery for legitimate traffic is needed to keep the services running?
5. **Delay in detecting and responding to the attack**: Time taken to initially detect attack and initiate mitigation response?
6. **False Positive Rate**: How much of legitimate traffic was flagged as attack traffic by the DDoS detection mechanism? The false positive rate for DDoS detection can be expressed as a function of legitimate background network traffic (*noise*).

7. **Legitimate services availability during the attack**: How many of the services are able to receive enough traffic to stay available?
8. **Amount of attack packets and number of hops traversed**: What is the DDoS collateral damage? How *close* to the attacker the defenses are able to block the attack?
9. **Impact of the DDoS attack on the network without any mitigation in place**: What was the impact of the mitigation algorithm on the DDoS attack?
10. **Application Throughput and Latency**: What was the impact of the DDoS attack on service/application quality of service while the DDoS mitigation algorithm was in effect? For example, VoIP calls, 4k Streaming Video, FTP.
11. **QoS service metrics**: What is the Mean Opinion Score time series distribution of the VoIP service?
12. **Infrastructure Switch Support**: How much of the infrastructure (what fraction of the switches) throughout the network needs to be instrumented to detect/mitigate against the DDoS attack? What is the optimal distribution (Benefit-Cost Analysis) of switches throughout the network?

Before we describe the evaluation of each attack scenario (UDP, HTTP, SYN), it is important to document the application-level metrics that are used to demonstrate the impact of DDoS attacks. For the VoIP application two metrics were calculated:

1. **The percentage of successful/failed calls**: how many calls were able to go through before, during and after the attack is mitigated.
2. **The Mean Opinion Score**: for the successful calls what is the perceived quality of calls before, during and after the attack is mitigated. Since we use the G.711 codec for the RTP (voice) streams, the maximum score that can be achieved is 4,4.

For the file transfer application, the transfer rate (unit is Mbps) was used to evaluate the impact of the DDoS attack. The DDoS Detector implemented cover three types of DDoS attacks that were identified as the main modes for the testing. These are the UDP flood attack (§5.1), the HTTP flood attack (§5.2), and the TCP SYN flood attack (§5.3).

5.1 UDP flood attack

The next four graphs show the attack timeline for the UDP flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third and fourth graph concern the file transfer application. We observe that in all graphs, the detection delay is 9 seconds. The detection delay is defined as the time between the first attack packet was sent and the time the detection algorithms flags the malicious UEs(User Equipments) for blocking. In the case of VoIP we observe that both the number of successful calls and Mean Opinion Score (MOS) is sustained during the first 8 seconds of the attack (this is due to the fact that some calls were already in progress and that the link is not immediately saturated) but then both metrics rapidly deteriorate. The number of successful calls drop to 1% while MOS drops to 0.9 during the attack. In the case of file transfer, the rate drops from 110Mbps down to 40Mbps during the attack.

The VoIP application recovers immediately after the attack is mitigated. We observe that successful call rate return to 100% and MOS goes back to 4,4 after the attack is mitigated (around second 45 of the timeline). The file transfer also recovers but it takes close to two and half minutes to go back to the original transfer rate before the attack takes place. This is due to how the TCP protocol works and how congestion is handled by the Linux kernel (secure copy protocol is TCP-based). Figure 5.4 shows the recovery progress of the file transfer after the attack. In all scenarios, the response time was maximum 16 seconds and the false positive rate is 0%.

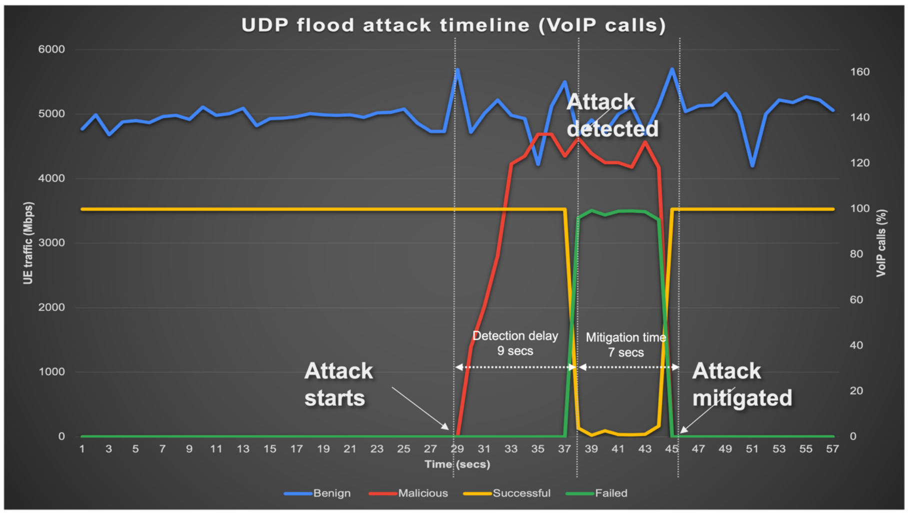


Figure 5.1: UDP flood attack timeline (VoIP calls)

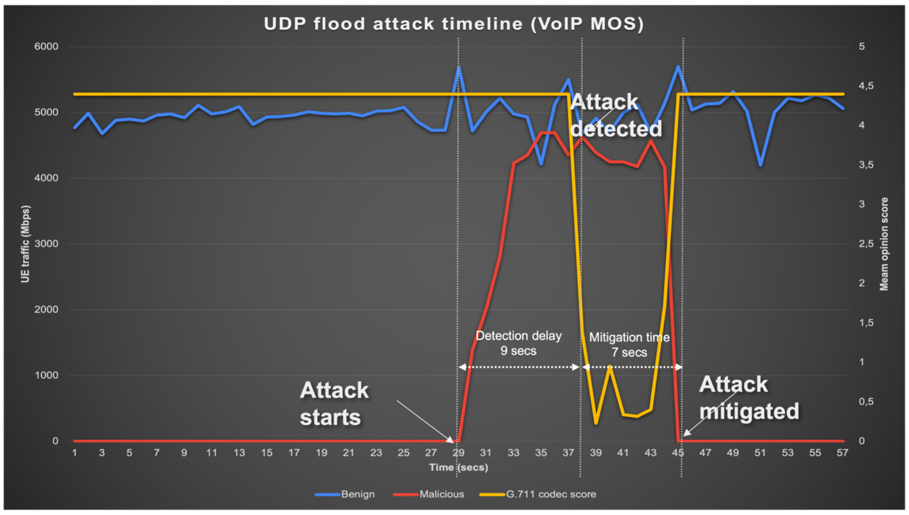


Figure 5.2: UDP flood attack timeline (SCP metrics)

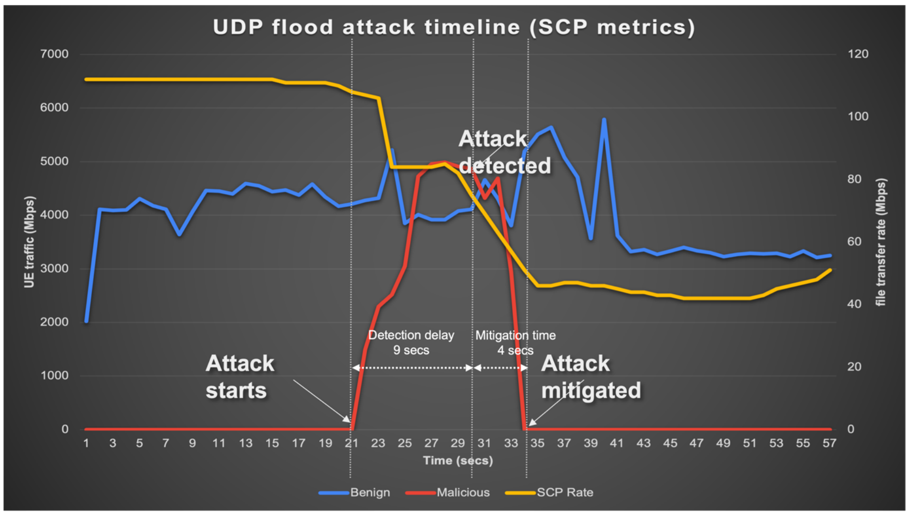


Figure 5.3: UDP flood attack timeline (VoIP calls)

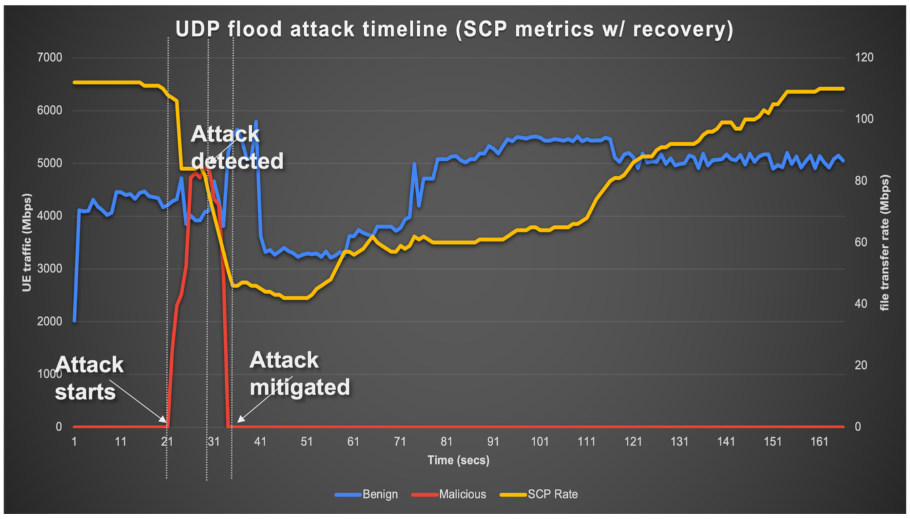


Figure 5.4: UDP flood attack timeline (SCP metrics) with recovery progress

The following table summarizes the metric results according to the 12 metrics mentioned at the start of this chapter.

Test Objective	Test Case / Critical Question	Result
False Positive Rate	How much of the legitimate traffic was flagged as attack traffic by the DDoS mitigation algorithm?	0%
Percentage of attack traffic dropped	How much of the attack traffic the DDoS mitigation algorithm was able to drop at various points – at the victim machine, at the edge and at the core in the network?	100%. All malicious IP addresses were throttled. All drops happened at the core switch of each node.
Delay in detecting and responding to the attack	How long did it take for the performer's detection mechanism to detect the DDoS attack?	In the file transfer scenario, the first attack packet to detection took 9 seconds. From detection to mitigation took 11 seconds. From the first attack packet to mitigation took in a total of 20 seconds. In the VoIP scenario, the first attack packet to detection took 18 seconds. From detection to mitigation took 10 seconds. From first attack packet to mitigation took in a total of 28 seconds.
What was the broad impact of the DDoS attack on the network without any mitigation in place?	What was the impact of the performer's mitigation algorithm on the DDoS attack?	Availability to the victim Web service was restored successfully and all malicious users were throttled. The number of failed called returned to 0.
Amount of attack packets and number of hops traversed	How close to the attacker the defenses were able to block the attack?	1 hop away

Legitimate traffic's goodput	How much of the good traffic was reaching the destination?	In terms of packets, all traffic from legitimate users was reaching the destination. In terms of application-level requests: before the attack begins 100% of the requests were served, during the attack 76% of the requests were served and after the mitigation takes place the percentage increases back to 100%
Delay and loss rate	What is the delay and loss rate of the legitimate packets?	The mean service delay before the attack begins was 12 ms, during the attack 498 ms and after the attack was throttled was 14ms
Application Throughput and Latency	What was the impact of the DDoS attack on service/ application quality of service while the DDoS mitigation algorithm was in effect? For example, VoIP calls, 4k Streaming Video, FTP	The file transfer rate drops from 110 Mbytes/sec down to 20 Mbytes/sec during the DDoS attack. After the mitigation takes place, the initial transfer rate is restored. The VoIP call rate drop from 100% success rate down to 30-50% success rate during the DDoS attack. After the mitigation takes place, the initial success rate of calls is restored.
QoS Service Metrics	What is the Q-Score/R-Score/Mean Opinion Score time series distribution of the VoIP service?	Before the attack, the Mean Opinion Score was 4,4 (the maximum for G.711 codec that was used in the RTP streams). During the attack. the score drops down between 1,5 to 2. After the attack, the score returns to 4,4.

Infrastructure Switch Support	How much of the infrastructure (what fraction of the switches) throughout the network needs to be instrumented to detect/ mitigate against the DDoS attack? What is the optimal distribution (Benefit-Cost Analysis) of switches throughout the network?	100(10 switch in our testbed)%

5.2 HTTP flood attack

The next three graphs show the attack timeline for the HTTP flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third graph concerns the file transfer application. In the case of VoIP we observe that both the number of successful calls and Mean Opinion Score (MOS) drop immediately once the attack starts. Successful calls drop to approximately 40% during the attack and MOS goes down to 1,5. In the case of file transfer, the rate drops from 110Mbps down to 25Mbps during the attack.

The VoIP application recovers immediately after the attack is mitigated. We observe that successful call rate return to 100% and MOS goes back to 4,4 after the attack is mitigated (around second 45 of the timeline for VoIP and second 35 for timeline of SCP). The file transfer also recovers immediately once the attack is mitigated. This is due to how the TCP protocol works and how congestion is handled by the Linux kernel (secure copy protocol is TCP-based). 5.7 shows the recovery progress of the file transfer after the attack.

In all scenarios, the response time was maximum 28 seconds and the false positive rate is 0%.

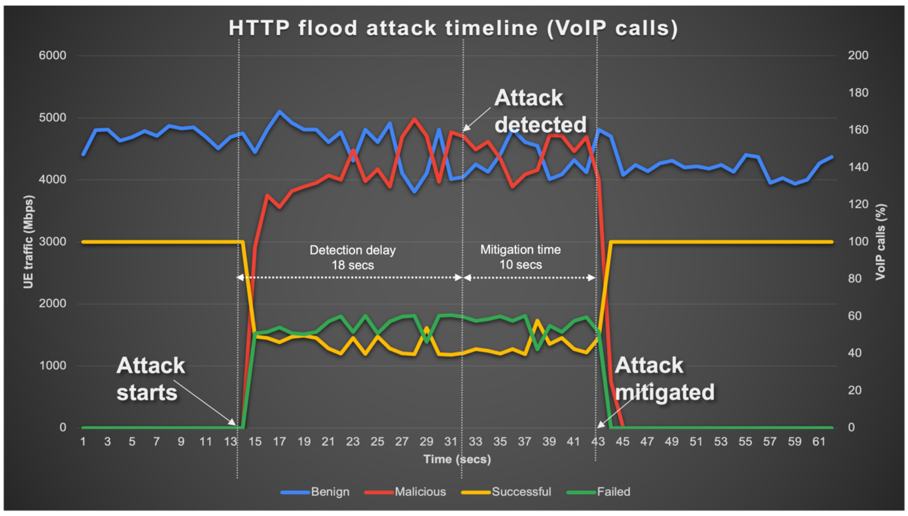


Figure 5.5: HTTP flood attack timeline (VoIP calls)

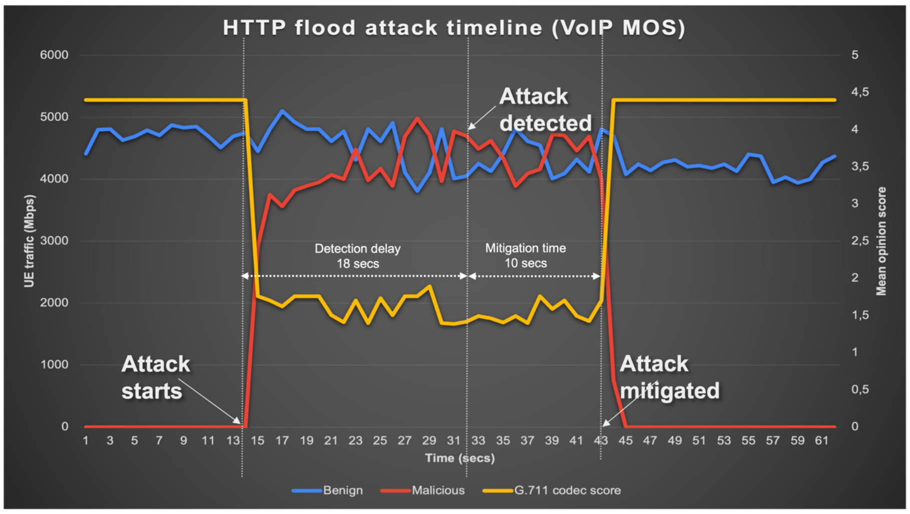


Figure 5.6: HTTP flood attack timeline (VoIP MOS)

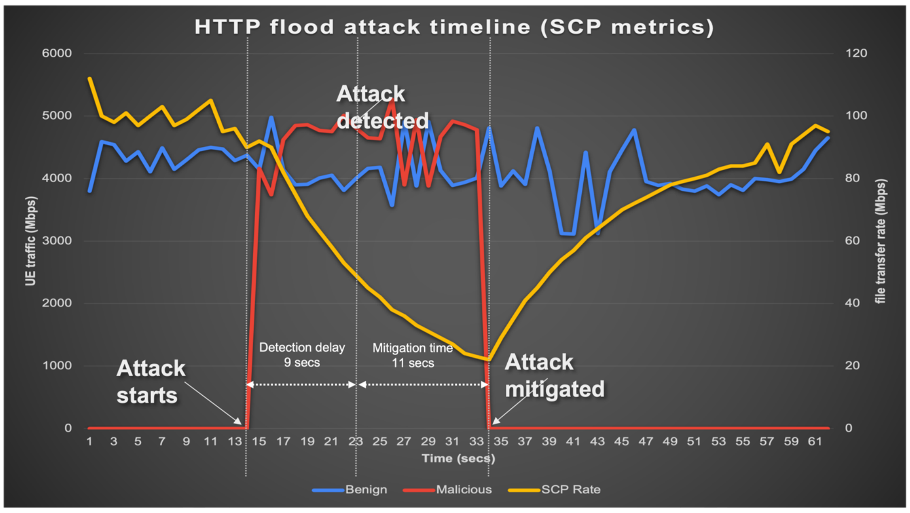


Figure 5.7: HTTP flood attack timeline (SCP metrics)

The following table summarizes the metric results according to the 12 metrics identified earlier.

Test Objective	Test Case / Critical Question	Result
False Positive Rate	How much of the legitimate traffic was flagged as attack traffic by the DDoS mitigation algorithm?	0%
Percentage of attack traffic dropped	How much of the attack traffic the DDoS mitigation algorithm was able to drop at various points – at the victim machine, at the edge and at the core in the network?	100%. All malicious IP addresses were throttled. All drops happened at the core switch of each node.
Delay in detecting and responding to the attack	How long did it take for the performer's detection mechanism to detect the DDoS attack?	The time between the first attack packet and the detection was 9 seconds. From detection to mitigation another 4 to 7 seconds were required. The total mitigation time is 13-16 seconds
What was the broad impact of the DDoS attack on the network without any mitigation in place?	What was the impact of the performer's mitigation algorithm on the DDoS attack?	Availability to the victim Web service was restored successfully and all malicious users were throttled. The number of failed called returned to 0.
Amount of attack packets and number of hops traversed	How close to the attacker the defenses were able to block the attack?	1 hop away

Legitimate traffic's goodput	How much of the good traffic was reaching the destination?	In terms of packets, all traffic from legitimate users was reaching the destination. In terms of application-level requests: before the attack begins 99.2% of the requests were served, during the attack 65% of the requests were served and after the mitigation takes place the percentage increases back to 100%
Delay and loss rate	What is the delay and loss rate of the legitimate packets?	The mean service delay before the attack begins was 98 ms, during the attack 494 ms and after the attack was throttled was 275ms. Given enough time the application restores to the original rate.
Application Throughput and Latency	What was the impact of the DDoS attack on service/ application quality of service while the DDoS mitigation algorithm was in effect? For example, VoIP calls, 4k Streaming Video, FTP	The file transfer rate drops from 110 Mbytes/sec down to 40 Mbytes/sec during the DDoS attack. After the mitigation takes place, the initial transfer rate is restored but after few minutes. The VoIP call rate drop from 100% success rate down to 4-10% success rate during the DDoS attack. After the mitigation takes place, the initial success rate of calls is restored.

QoS Service Metrics	What is the Q-Score/R-Score/Mean Opinion Score time series distribution of the VoIP service?	Before the attack, the Mean Opinion Score was 4,4 (the maximum for the G.711 codec used in the RTP streams). During the attack, the score drops between 0,5 and 1. After the attack is mitigated, the score returns to 4,4.
Infrastructure Switch Support	How much of the infrastructure (what fraction of the switches) throughout the network needs to be instrumented to detect/ mitigate against the DDoS attack? What is the optimal distribution (Benefit-Cost Analysis) of switches throughout the network?	100 (10 switch in our testbed)%

5.3 SYN flood attack

The next three graphs show the attack timeline for the SYN flood attack. The first two graphs display the VoIP metrics on the secondary Y-axis while the third graph concerns the file transfer application. The SYN flood attack has insignificant impact on both applications. The reasons are threefold. First, the impact of a SYN flood attack on the bandwidth is minimal. Since SYN packets are very small (close to 60 bytes), even a large number of SYN packets per second cannot congest a link. The total amount of traffic generated is close to 100Mbps (as a reminder the link is 10Gbps). Secondly, the SYN flood attacks an HTTP server so it has no material impact on the SIP service which is UDP-based or the file transfer since we do not attack the used services. Finally, the targeted host has SYN-cookies implemented by default (and most Linux distributions out of the box nowadays) so the attack cannot deplete resources on the target machine. SYN cookies prevent the host of allocating file descriptors unless the UE completes the TCP handshake. In all scenarios, the response time was maximum 14 seconds and the false positive rate is 0%.

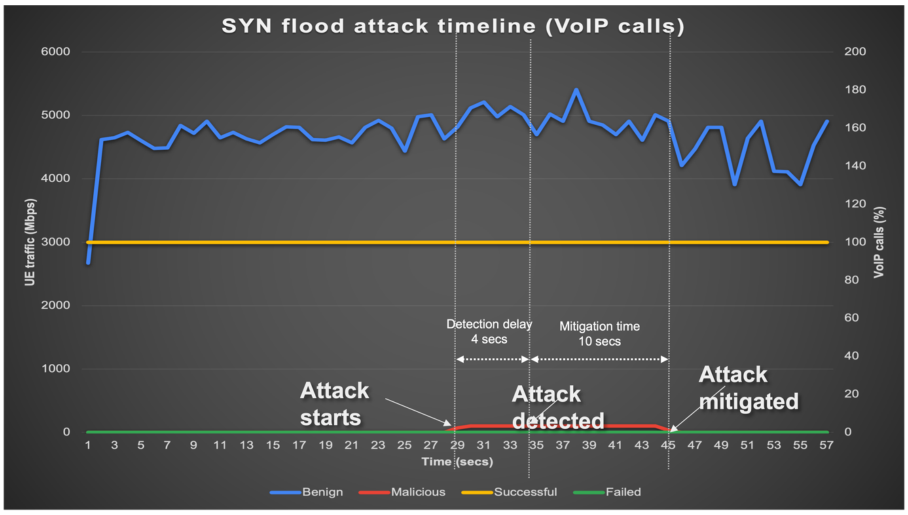


Figure 5.8: SYN flood attack timeline (VoIP calls)

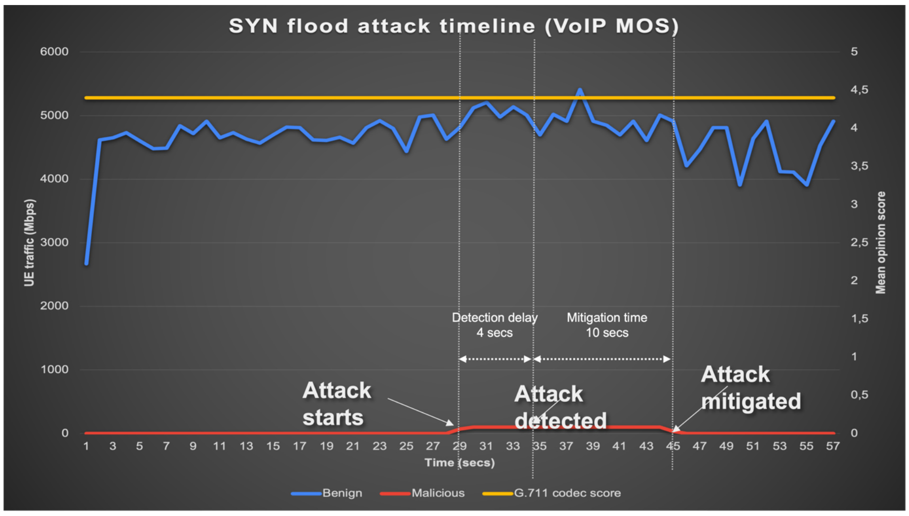


Figure 5.9: SYN flood attack timeline (VoIP MOS)

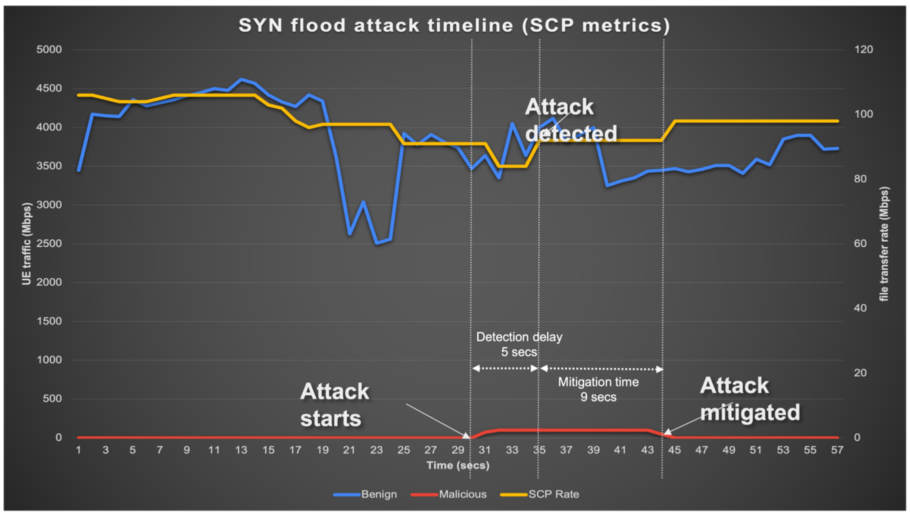


Figure 5.10: SYN flood attack timeline (SCP metrics)

The following table summarizes the metric results according to the 12 metrics identified earlier.

Test Objective	Test Case / Critical Question	Result
False Positive Rate	How much of the legitimate traffic was flagged as attack traffic by the DDoS mitigation algorithm?	0%
Percentage of attack traffic dropped	How much of the attack traffic the DDoS mitigation algorithm was able to drop at various points – at the victim machine, at the edge and at the core in the network?	100%. All malicious IP addresses were throttled. All drops happened at the core switch of each node.
Delay in detecting and responding to the attack	How long did it take for the performer's detection mechanism to detect the DDoS attack?	The time between the first attack packet and the attack detection was 4-5 seconds. From detection to mitigation another 9-10 seconds were required. From the first attack packet to mitigation the maximum observed reaction time was 14 seconds.
What was the broad impact of the DDoS attack on the network without any mitigation in place?	What was the impact of the performer's mitigation algorithm on the DDoS attack?	Availability to the victim Web service was restored successfully and all malicious users were throttled.
Amount of attack packets and number of hops traversed	How close to the attacker the defenses were able to block the attack?	1 hop away

Legitimate traffic's goodput	How much of the good traffic was reaching the destination?	In terms of packets, all traffic from legitimate users was reaching the destination. In terms of application-level requests: before the attack begins 100% of the requests were served, during the attack 0% of the requests were served (since server queue was full) and after the mitigation takes place the percentage increases back to 100%.
Delay and loss rate	What is the delay and loss rate of the legitimate packets?	The mean service delay before the attack begins was 48 ms, during the attack 98 ms and after the attack was throttled was 49ms.
Application Throughput and Latency	What was the impact of the DDoS attack on service/ application quality of service while the DDoS mitigation algorithm was in effect? For example, VoIP calls, 4k Streaming Video, FTP	The file transfer rate before the attack was 100-110 Mbytes/sec and there was small rate drop during the DDoS attack of about 10Mbps. This is expected since the SYN flood attack does not consume considerable bandwidth. The VoIP call rate had 100% success rate during the DDoS attack. This is expected since the SYN flood attack does not consume considerable bandwidth and the SIP servers were not targeted.
QoS Service Metrics	What is the Q-Score/R-Score/Mean Opinion Score time series distribution of the VoIP service?	The Mean Opinion Score was 4,4 throughout the entire time.

Infrastructure Switch Support	How much of the infrastructure (what fraction of the switches) throughout the network needs to be instrumented to detect/ mitigate against the DDoS attack? What is the optimal distribution (Benefit-Cost Analysis) of switches throughout the network?	100 (10 switch in our testbed)%

Chapter 6

Related work

Here is presented some similar work, already done that motivated the PRINCIPALS project and this thesis.

A Secure Active Network Environment Architecture (AN) [39] : The whole concept of the Active Networks (AN) is a foundation for the PRINCIPALS project. PRINCIPALS is based in the idea of a network that can dynamically change its configuration and its capabilities. Active Networks is the passageway to a secure network that can also introduce safe programmability and adaptability. PRINCIPALS will build upon the inovative work done in the DARPA Active Networks program, the intellectual ancestor to Software Defined Networks (SDNs) [45].

The above work alongside the continuation by J.T. Moore in the **Safe and Efficient Active Packets** [48] project, that introduced us with the concept of Active Packets and the specifically designed language of SNAP(Safe Networking with Active Packets), are the cornerstones of PRINCIPALS. PRINCIPALS will be built upon the lessons learnt from previous work and will try to expand their functionality.

A Secure Plan (Extended Version) [**43**]: The implementation of TAMElets, the "Thin" AMElets, which has already been discussed above, in the Framework chapter, is "inspired" from PLAN. Borrowing elements from the PLAN project and extending them will be an initial approach to the TAMElet implementation. We anticipate TAMElets will be expressed in a similar programming language, and will implement the necessary traceroute-like functionality.

PLAN was one of the first active networking systems, and the first to be demonstrated at the DARPA active networks workshops. It was also the first packet-scripting language and the first to use language-based resource restrictions to limit the impact of an active packet on the network. PLAN has received the most formal analysis of any active networking system, including an abstract calculus, mathematical specification, formal simulations and theorems about invariants and properties of applications. The PLAN system was also the foundation or target of a number of studies of security, including trust management, information flow and active firewalls. PLAN introduced the idea of implementing network layers through the use of a quotation-like mechanism (chunks).It was also the first AN system to prove that it could implement Internet (IP-based) functionality (as part of the PLANet system) and then improve upon it to support Multicast and Packet-directed routing (FBAR) among other applications.

Flexible Network Monitoring with FLAME [40] : FLAME is a project built on Active Networks and provides an open architecture for network traffic monitoring. PRINCIPALS will use the FLAME architecture as an initial approach for the FAMElets. PRINCIPALS contributors had a fair share of contribution in the FLAME project and they plan to not only leverage elements from the project but extend them as well.

xPF: Packet Filtering for Low-Cost Network Monitoring [44] : xPF constitutes one more project that PRINCIPALS contributors have worked on and are gonna exploit. xPF is a high-performance Turing-complete packet filtering language with resource constraints, we will implement upper bounds to the execution time of TAMElets. It's similar to the BPF language with the main difference being that it allows backward jumps.

The KeyNote Trust-Management System, Version 2 [41] : KeyNote is also one of the projects that PRINCIPALS contributors have worked on and are gonna exploit. The KeyNote architecture and language are useful as building blocks for the trust management aspects of a variety of Internet protocols and services. s a starting point, KeyNote is going to be used as a trust management system for authorization credentials in PRINCIPALS. This credentials will contain information about the resource constraints and permissions of the AMElets (TAMElets or FAMElets), as it has already been explained in Chapter 3.

WebSOS [49]: Another novel architecture from contributors of PRINCIPALS, implemented in the past, can be used in this project as well. WebSOS is a novel overlay-based architecture that provides guaranteed access to a web server that is targeted by a denial of service (DoS) attack. In a concept of a DDoS attack, PRINCIPALS could implement an appropriate DCO to deal with.

A quick example. In this scenario a DDoS attack has already been detected. As a first step, the network SOC broadcasts TAMElets that create a reserved management slice within the base 5G network, allowing for telemetry to be collected by any network element. Subsequent TAMElets configure the network switches to report netflow for non-established TCP flows. Initial triage of this data suggests that the bulk of the attack traffic is coming from external IP sources from around the world using a variety of random traffic. Quickly, new TAMElets are emitted that configure SDN switches to prioritize existing flows, limiting other traffic to a fraction of the available bandwidth. In parallel, FAMElets implementing active TCP SYN cookies [29] and CAPTCHA-based IP validation 5 are instantiated in

5G nodes near the network ingress points. Within a minute, this blunts the force of the DDoS attack, and services begin to slowly become responsive.

Chapter 7

Conclusion

Nowadays mobile communications are a part of our daily life. The growth of this department and the continuation of that growth is certain. At the same time, modern networking has become a constant battlefield between hackers and cybersecurity operators. Since the launch of 5G, the attackers are able to make their moves in a much wider surface. Malicious users will launch attacks in any internet facing system that shows vulnerabilities. This growth in aspects of range and rate of these attacks showed that the former naive stationary approaches on computer security are no longer viable. A new *mobile* way to defend is required, in order to be a step ahead of the attackers. PRINCIPALS introduces a novel architecture for safe programmability and adaptability in the 5G network. Through the use of Active Networks, PRINCIPALS will give the ability to reconfigure, adapt and overcome any possible attack in any possible networking space. PRINCIPALS provides a novel mechanism for conducting defensive cybersecurity operations at a scale, pace, and precision that is unprecedented. In this thesis we presented:

- A series of network security primitives that PRINCIPALS will use, that will provide the ability to defend against malicious users trying to take advantage of the 5G infrastructure. These primitives will use containerised code, tested and evaluated in this thesis as well. The containerised code will be deployed inside the PRINCIPALS environment and will address current and anticipated network security problems.
- Beside the network security primitives, PRINCIPALS also provides the ability of *mobile* DCOs, with code-carrying packets with limited permissions, that will secure adaptability and flexibility in this environment. By extracting the set of permission that the network security primitives require, we set an upper-bound limit on what actions the code-carrying packets can take.

In this way, the defensive cyber operators will have the ability to match the speed, scale, and accuracy of attacker tools, without introducing new points of vulnerability or risk network instability.

Bibliography

[1] 3gpp. https://www.3gpp.org, the 3rd Generation Partnership Project is an umbrella term for a number of standards organizations which develop protocols for mobile telecommunications.

[2] 5g roll–out in the eu: delays in deployment of networks with security issues remaining unresolved. https://op.europa.eu/webpub/eca/special-reports/security-5g-networks-03-2022/en/, while 5G has the potential to unleash many opportunities for growth, it comes with certain risks.

[3] A. sparling et al. (2018) event-stream. https://github.com/dominictarr/event-stream/issues/116, i don't know what to say.

[4] Antrea. https://antrea.io, antrea is a Kubernetes-native project that implements the Container Network Interface (CNI) and Kubernetes NetworkPolicy.

[5] beamforming. https://spectrum.ieee.org/5g-bytes-beamforming-explained, beamforming is a type of radio frequency (RF) management in which a wireless signal is directed toward a specific receiving device.

[6] Bitcoin. https://bitcoin.org/en, bitcoin is a decentralized digital currency that can be transferred on the peer-to-peer bitcoin network. Bitcoin transactions are verified by network nodes through cryptography and recorded in a public distributed ledger called a blockchain.

[7] Containers' security. https://snyk.io/learn/container-security/, what is container security?

[8] Containers vs vms? https://www.atlassian.com/microservices/cloud-computing/containers-vs-vms, containers vs VMs. What are the differences?

[9] Containers vs vms? https://www.netapp.com/blog/containers-vs-vms/, containers vs VMs. What are the differences

[10] Docker. https://www.docker.com, docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers.

[11] Docker hub. https://hub.docker.com, docker hub is the world's easiest way to create, manage, and deliver your team's container applications.

[12] ecpri. https://wiki.wireshark.org/eCPRI.md, enhanced CPRI (eCPRI) is a way of splitting up the baseband functions to reduce traffic strain on the fiber.

[13] Evolution of internet of things. https://www.techaheadcorp.com/knowledge-center/evolution-of-iot/, evolution of Internet of Things.

[14] Evolution of networking generations. https://www.brainbridge.be/en/blog/1g-5g-brief-history-evolution-mobile-standards, evolution of Networking Generations.

[15] Github. https://github.com, gitHub, Inc., is an Internet hosting service for software development and version control using Git.

[16] Keras. https://keras.io, keras is an open-source software library that provides a Python interface for artificial neural networks.

[17] Kubernetes. http://kubernetes.io, kubernetes, also known as K8s, is an open-source system for automating deployment, scaling, and management of containerized applications.

[18] Mec. https://www.etsi.org/technologies/multi-access-edge-computing, multi-access edge computing (MEC) is a type of network architecture that provides cloud computing capabilities and an IT service environment at the edge of the network.

[19] Mininet. http://mininet.org/, mininet creates a realistic virtual network, running real kernel, switch and application code, on a single machine (VM, cloud or native), in seconds, with a single command.

[20] Multus. https://github.com/k8snetworkplumbingwg/multus-cni, multus CNI is a container network interface (CNI) plugin for Kubernetes that enables attaching multiple network interfaces to pods.

[21] npm. https://www.npmjs.com, npm is a package manager for the JavaScript programming language maintained by npm, Inc. npm is the default package manager for the JavaScript runtime environment Node.js.

[22] npm, inc. (2018). https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident, details about the event-stream incident.

[23] O-ran. https://www.o-ran.org, an Open Radio Access Network (O-RAN) is a totally disaggregated approach to deploying mobile fronthaul and midhaul networks built entirely on cloud native principles.

[24] ovs. openvswitch.org/, open vSwitch, sometimes abbreviated as OVS, is an open-source implementation of a distributed virtual multilayer switch.

[25] S. yegulalp (2016). http://www.infoworld.com/article/3047177/javascript/how-one-yanked-javascript-package-wreaked-havoc.html., how one yanked javascript package wreaked havoc.

[26] Scapy. https://scapy.net, scapy is a packet manipulation tool for computer networks, originally written in Python by Philippe Biondi. It can forge or decode packets, send them on the wire, capture them, and match requests and replies.

[27] Scapy usage. https://scapy.readthedocs.io/en/latest/usage.html, more detailed usage of Scapy's Sniff() function.

[28] Snyk. (2016). https://snyk.io/, find, fix and monitor for known vulnerabilities in node.js and ruby packages.

[29] Syncookies. https://cr.yp.to/syncookies.html, sYN cookies are particular choices of initial TCP sequence numbers by TCP servers.

[30] Top eight virtualization security issues and risks. https://www.liquidweb.com/kb/virtualization-security-issues-and-risks/, vMs security risks.

[31] What is 5g security? explaining the security benefits and vulnerabilities of 5g architecture. https://cybersecurity.att.com/blogs/security-essentials/what-is-5g-security, explaining the security benefits and vulnerabilities of 5G architecture.

[32] What is 5g vm? https://www.vmware.com/topics/glossary/content/virtual-machine.html, explaining what a Virtual Machine is.

[33] What is a container? https://www.docker.com/resources/what-container/, what is a container?

[34] What is a container image? https://www.aquasec.com/cloud-native-academy/container-security/container-images/, what is a container image?

[35] What is mirai botnet? https://www.cloudflare.com/learning/ddos/glossary/mirai-botnet/, what is Mirai botnet?

[36] What is nfv? https://www.vmware.com/topics/glossary/content/network-functions-virtualization-nfv.html, what is NFV?

[37] What is vm hypervisor? https://www.howtogeek.com/66734/htg-explains-what-is-a-hypervisor/, explaining what a Virtual Machine Hypervisor is.

[38] Whats is container security? https://www.trendmicro.com/en_us/what-is/container-security.html, the process of securing containers is continuous.

[39] Alexander, D.S., Arbaugh, W.A., Keromytis, A.D., Smith, J.M.: A secure active network environment architecture: realization in switchware. IEEE network **12**(3), 37–45 (1998)

[40] Anagnostakis, K.G., Greenwald, M.B., Ioannidis, S., Li, D., Smith, J.M.: Flexible network monitoring with flame. Computer Networks **50**(14), 2548–2563 (2006)

[41] Blaze, M., Feigenbaum, J., Ioannidis, J., Keromytis, A.: The keynote trust-management system version 2. Tech. rep., - (1999)

[42] Cadariu, M., Bouwers, E., Visser, J., van Deursen, A.: Tracking known security vulnerabilities in proprietary software systems. In: 2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER). pp. 516–519. IEEE (2015)

[43] Hicks, M., Keromytis, A.D., Smith, J.M.: A secure plan (extended version). In: Proceedings DARPA Active Networks Conference and Exposition. pp. 224–237. IEEE (2002)

[44] Ioannidis, S., Anagnostakis, K.G., Ioannidis, J., Keromytis, A.D.: xpf: packet filtering for low-cost network monitoring. In: Workshop on High Performance Switching and Routing, Merging Optical and IP Technologie. pp. 116–120. IEEE (2002)

[45] Kreutz, D., Ramos, F.M., Verissimo, P.E., Rothenberg, C.E., Azodolmolky, S., Uhlig, S.: Software-defined networking: A comprehensive survey. Proceedings of the IEEE **103**(1), 14–76 (2014)

[46] Lauinger, T., Chaabane, A., Arshad, S., Robertson, W., Wilson, C., Kirda, E.: Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web. arXiv preprint arXiv:1811.00918 (2018)

[47] Maass, M.: A theory and tools for applying sandboxes effectively. - (2016)

[48] Moore, J.T.: Safe and efficient active packets. - (1999)

[49] Morein, W.G., Stavrou, A., Cook, D.L., Keromytis, A.D., Misra, V., Rubenstein, D.: Using graphic turing tests to counter automated ddos attacks against web servers. In: Proceedings of the 10th ACM conference on Computer and communications security. pp. 8–19 (2003)

[50] Staicu, C.A., Pradel, M., Livshits, B.: Understanding and automatically preventing injection attacks on node. js. Tech. Rep. TUD-CS-2016-14663, TU Darmstadt, Department of Computer Science, Tech. Rep. (2016)

[51] Vasilakis, N., Ntousakis, G., Heller, V., Rinard, M.C.: Efficient module-level dynamic analysis for dynamic languages with module recontextualization. In: Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 1202–1213 (2021)

[52] Vasilakis, N., Staicu, C.A., Ntousakis, G., Kallas, K., Karel, B., DeHon, A., Pradel, M.: Mir: Automated quantifiable privilege reduction against dynamic library compromise in javascript. arXiv preprint arXiv:2011.00253 (2020)

[53] Zimmermann, M., Staicu, C.A., Tenny, C., Pradel, M.: Small world with high risks: A study of security threats in the npm ecosystem. In: 28th USENIX Security Symposium (USENIX Security 19). pp. 995–1010 (2019)

Appendices

Appendix A

Creating Dockerised Applications

Here is a guide on how to dockerise an application. As we have already seen in this thesis, the Dockerisation of an application is a need in every single network security primitive implemented. So let's see how to use Docker in order to containerise an app.

The very first step is to install Docker on your PC or the server you are working on. Docker is going to be used for the containerisation. Having Docker installed, you are now going to write your Dockerfile. The Dockerfile is a script for building a docker image. Inside it, there specific instructions to set up the image and install all the necessary requirements. Let's now take a deeper look of the Dockerfile, by using as an example the Dockerfile i used for the creation of the Snort Primitive. Below is the Dockerfile:

```
FROM ubuntu:20.04


ENV DEBIAN_FRONTEND noninteractive
ENV NETWORK_INTERFACE eth0

RUN apt-get update && apt-get -y install \
    wget \
    build-essential \
    gcc \
    libpcre3-dev \
    zlib1g-dev \
    libluajit-5.1-dev \
    libpcap-dev \
    openssl \
    libssl-dev \
    libnghttp2-dev \
    libdumbnet-dev \
    bison \
    flex \
    libdnet \
    autoconf \
    libtool \
    nodejs \
    tcpdump \
    npm
```

```

WORKDIR /opt

ENV DAQ_VERSION 2.0.7
RUN wget https://www.snort.org/downloads/snort/daq-${DAQ_VERSION}.tar.gz \
    && tar xvfz daq-${DAQ_VERSION}.tar.gz \
    && cd daq-${DAQ_VERSION} \
    && ./configure; make; make install

ENV SNORT_VERSION 2.9.18.1
RUN wget https://www.snort.org/downloads/archive/snort/snort-${SNORT_VERSION}.tar.gz \
    && tar xvfz snort-${SNORT_VERSION}.tar.gz \
    && ls \
    && cd snort-${SNORT_VERSION} \
    && ./configure; make; make install

RUN ldconfig

ADD mysnortrules /opt
RUN mkdir -p /var/log/snort && \
    mkdir -p /usr/local/lib/snort_dynamicrules && \
    mkdir -p /etc/snort && \
    cp -r /opt/rules /etc/snort/rules && \
    mkdir -p /etc/snort/preproc_rules && \
    mkdir -p /etc/snort/so_rules && \
    cp -r /opt/etc /etc/snort/etc && \
    touch /etc/snort/rules/white_list.rules /etc/snort/rules/black_list.rules

RUN apt-get clean && rm -rf /var/lib/apt/lists/* /tmp/* /var/tmp/* \
    /opt/snort-${SNORT_VERSION}.tar.gz /opt/daq-${DAQ_VERSION}.tar.gz

COPY myjsscript.js /var/log/snort/myjsscript.js
RUN chmod +x /var/log/snort/myjsscript.js
```

Starting from top, the FROM command is used to set the base image of your image. In the example above, as you can see, we take the base image of ubuntu:20.04 also known as Focal Fossa. This is going to be the OS of our app.

Right after, the ENV command is used. The ENV command is used in order to set the value of a variable. For example, ENV DEBIAN_FRONTEND noninteractive is used in order to set the environment to noninteractive, in other words that there will be no dialogue during apt-get, etc.

The 'RUN' instruction will run a command. Here you can see that RUN is used in order to install all the necessary libraries and packages like npm , nodejs etc. These are all absolute necessities for the app that need to be installed.

The 'WORKDIR' instruction will set the default working directory that we will use to run the application. In our case, this directory is the /opt directory.

Last but not list in our Dockerfile, you can see the 'ADD' and the 'COPY' command. With the add command we basically copy a file or a directory from our local host to the directory we desire inside the Docker image while the COPY command will do the same but in the Docker container. There are a lot of commands to be used like 'EXPOSE', 'CMD', etc . For more details you can visit the official Docker website.

The Dockerfile should be placed inside the directory you're going to use, let's name it startDir. Inside the startDir place every file you are going to need and add in your Dockerfile. When all is done, you are ready to build your container. Open a terminal and go to your app directory, the startDir. Then execute the following command:

```
docker build -t mycontainer .
```

The process of building your container will begin, and if no errors occur, your container will shortly be ready. The -t command allows you to name the container in the way you want, in our case "mycontainer". Note that it is necessary to be inside the directory in order to use the docker build command. Now that your containerized application is ready, you can easily run it using the command:

```
docker run mycontainer .
```

For more information about the arguments you can use to run your application with, visit the official site of Docker.

Appendix B

How to monitor network traffic using Scapy

Here is a guide on how to monitor network traffic. As we have already seen in this thesis, the monitoring of traffic is a very common task and is used in almost every primitive implemented. In this appendix we show a guide on how to do this, using Python and Scapy [26].

The very first step is to install Python and Scapy on your PC or the server you are working on. The installation is a pretty standard procedure so we are not going to further analyse that. When the installation is done, you are now going to start writing your Python script. Below we will show and discuss the script used for the necessary functionality of the Flow Recording Primitive. Here is the Python script:

```python
from scapy.all import *
import socket
import argparse
import time
import json

class DomainList:
    domains = None

    def exist(self, domain):
        if domain in self.domains:
            return True
        else:
            return False
        pass

    def load_domains(self, path):
        self.domains = set(line.strip() for line in open(path))
        pass

    def __init__(self, path):
        self.load_domains(path)


class PacketInfo(object):

    def create_send_object(self):
        obj = {
            "ts": self.ts,
            "domain": self.domain,
            "ip_src": self.ip_src,
            "ip_dst": self.ip_dst,
            "resolved_ip": self.resolved_ip,
            "mac_src": self.mac_src,
            "mac_dst": self.mac_dst,
            "port_src": self.port_src,
            "port_dst": self.port_dst
        }
        return json.dumps(obj)

    def __init__(self, ts, domain, ip_src, ip_dst, resolved_ip, mac_src,
        mac_dst, port_src, port_dst):
        self.ts = ts
        self.domain = domain
        self.ip_src = ip_src
        self.ip_dst = ip_dst
        self.resolved_ip = resolved_ip
        self.mac_src = mac_src
        self.mac_dst = mac_dst
        self.port_src = port_src
        self.port_dst = port_dst

class PacketMonitor:
    iface = None
    detector = None

    def establish_connection(self, host="192.168.1.204", port=8080):
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.socket.connect((host, port))
        pass

    def send_data(self, data):
        self.arguments["data"] = data
        obj = {
            "action": self.action,
            "argument": self.arguments,
        }
        # Format json object
        send_obj = json.dumps(obj) + "\n"
        self.socket.sendall(send_obj.encode("utf-8"))
        pass

    def trim_domain_string(self, domain):
        domain = domain[:-2]
        domain = domain[2:]
        return domain

    def process_packet(self, packet):
        """
        This function is executed whenever a packet is sniffed
        """
        if IP not in packet:
            return

        if not packet.haslayer(DNS):
            return

        domains = None
        dns_layer = packet.getlayer(DNS)

        if dns_layer.ancount > 0 and dns_layer.qd:
            ts = time.time()
            ip_src = str(packet[IP].src)
            ip_dst = str(packet[IP].dst)
            mac_src = str(packet.src)
            mac_dst = str(packet.dst)
            if UDP in packet:
                port_src = str(packet[UDP].sport)
                port_dst = str(packet[UDP].dport)
            if TCP in packet:
                port_src = str(packet[TCP].sport)
                port_dst = str(packet[TCP].dport)

            domain = self.trim_domain_string(str(dns_layer.qd.qname))
            dga_result = self.domains.exist(domain)

            if dga_result == False:
                return

            for x in range(dns_layer.ancount):
                resolved_ip = str(dns_layer.an[x].rdata)
                packet_info = PacketInfo(ts, domain, ip_src, ip_dst,
                resolved_ip, mac_src, mac_dst, port_src, port_dst)
                send_object = packet_info.create_send_object()
                self.send_data(send_object)

    def sniff_packets(self):
        if iface:
            # `process_packet` is the callback
            sniff(filter="port 53", prn=self.process_packet, iface=iface, store=False)
        else:
            # sniff with default interface
            sniff(filter="port 53", prn=self.process_packet, store=False)

    def __init__(self, iface, domains, address, port, action, arguments):
        self.iface = iface
        self.domains = domains
        self.establish_connection(address, port)
        self.action = action
        self.arguments = arguments
        pass

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="DGA detector")
    parser.add_argument("-i", "--iface")
    parser.add_argument("-d", "--domains")
    parser.add_argument("-a", "--address")
    parser.add_argument("-p", "--port")
    parser.add_argument("-c", "--command", required=True)
    parser.add_argument("-arg", "--arguments", required=False)
    args = parser.parse_args()

    domains_file = args.domains
    domains = DomainList(domains_file)

    iface = args.iface
    address = args.address
    port = int(args.port)

```

```
150     action = args.command
151     if args.arguments == None:
152         arguments_json = "\{\}"
153     else:
154         arguments_json = args.arguments
155 
156     try:
157         arguments = json.loads(arguments_json)
158     except ValueError:
159         print("Decoding JSON has failed")
160         arguments = {}
161         pass
162 
163     packet_monitor = PacketMonitor(iface, domains, address, port, action, arguments)
164     packet_monitor.sniff_packets()
165     packet_monitor.socket.close()
```

As you can probably see, a lot are taking place inside this script. So let's see on which functions we are going to take a deeper dive to. We start with the necessary importation of Scapy, as you can see in line 1. Now inside the PacketMonitor class, we are interested about two functions, the `sniff_packets()` and the `process_packet()` functions. `sniff_packets()` is the function that provides the main functionality. As you can see, it is called from the "main", once the PacketMonitor class has been initialised. `sniff_packets()` checks the "iface" variable, the variable we use to store the input networking interface we want to monitor. If "iface" is not Null, we proceed to use the scapy function, `sniff()`, to monitor the interface we want. If it is Null, then we monitor the default Scapy interface. `sniff()` function monitors the networking traffic of the interface we define for an infinite period of time until the user interrupts it.

Another argument to define the functionality of `sniff()` is the "filter" argument. Depending on the value of the "filter" argument, `sniff()` is going to monitor only specific traffic. For example, as you can see in line 118, by giving the value "port 53" to "filter", we monitor only the DNS requests, since this is what we want in this Primitive.

`process_packet()` is the callback function that `sniff()` uses every time a packet is sniffed. This is realized by giving the function as value to the argument "prn" on the `sniff()` function. Inside the `process_packet()`, we provide the functionality we want. In this example, once we sniff a packet, we check if the domain matches one of the domains we want to monitor. If it does, we store the information we want about this packet and later send it to another pod. But that is obviously is going to differ from script to script. You can provide your own functionality inside your code, according to your needs. There are multiple ways `sniff()` can be used and a number of different arguments to be used with. For more details you can use the official site of Scapy [26], or see the manual [27].