Kubernetes for the Deployment of Mobile Core Networks

Cameron MacLeod
S1528565

4th Year Project Report
Computer Science
School of Informatics
University of Edinburgh

2020

// Abstract

Cloud-native network functions (CNFs) can significantly reduce operating costs for mobile operators, not just through their ability to be hosted in public clouds, but also since their enabling technology, containers, use compute resources more efficiently than alternatives. Kubernetes is the biggest technology in container orchestration and has the potential to bring many benefits to mobile networks, however, the existing work that looks at containerising mobile network functions and orchestrating them with Kubernetes does not make good use of the tools that it provides for free. In this project, I present the design, partial implementation and evaluation of 4GKube for orchestrating the 4G core network (EPC) with Kubernetes. I also present two novel attacks against Kubernetes and demonstrate that they won't work in the context of 4GKube.

Acknowledgements

Thanks to my supervisor for not giving up on me and for keeping me motivated and excited about the project. Thanks also to Jon, Rupen and the rest of the group for their help and for always being available. Thanks to my family and girlfriend for the moral support and understanding.

Table of Contents

1 Introduction **1**

2 Background **3**
2.1 Mobile Networks 3
2.1.1 Evolved Packet Core in 4G 4
2.1.2 4G and 5G 5
2.1.3 Virtualised and Cloud-native network functions 5
2.2 Kubernetes 5
2.2.1 Kubernetes concepts 5
2.3 Related work 7

3 4GKube Design **8**
3.1 Scope 8
3.1.1 Focus on the Core network 8
3.1.2 Reasoning for using EPC 8
3.1.3 Reasoning for using Kubernetes 9
3.2 Challenges 9
3.3 Goals 10
3.4 Architecture 11
3.5 Changes to the MME 12
3.6 Generalisation to 5GC 13

4 Implementation **14**
4.1 Pre-existing work 14
4.1.1 Powder 14
4.1.2 EPC implementations 15
4.1.3 openair-k8s 15
4.2 Creating a Powder profile 15
4.2.1 Existing profile 15
4.2.2 Creating a profile 15
4.2.3 Kubernetes cluster 17
4.3 Deploying OpenAirInterface on Powder 17
4.3.1 Building container images 18
4.3.2 Configuration changes 18
4.3.3 Deployment 18
4.4 Verifying deployment 19

4.5 Modification of MME 19

5 Evaluation **21**
5.1 Evaluation of design 21
5.1.1 Goals 21
5.1.2 Limitations 22
5.2 Evaluation of implementation 22
5.2.1 Functional tests 22
5.2.2 Performance tests 22

6 Security **24**
6.1 Kubernetes security 24
6.2 Attack Background 25
6.3 Attacking the replication controller 26
6.3.1 Motivation 26
6.3.2 Attack 26
6.3.3 Evaluation 27
6.4 Attacking the HPA controller 28
6.4.1 Motivation 28
6.4.2 Attack 28
6.4.3 Experiments 29
6.4.4 Evaluation 31

7 Conclusion **32**
7.1 Conclusion 32
7.2 Future work 32

Bibliography **33**

Chapter 1

Introduction

The state of the art in networking is moving from virtual network functions (VNFs) to cloud-native network functions (CNFs), a new paradigm relying on the concept of containers. These bring increased operational savings over their predecessors through more efficient use of compute resources and easier deployment. Mobile network operators are keen to tap into the economies this trend brings as evidenced through a growing number of efforts to containerise mobile network components [40, 53, 4, 5].

One area where containerisation could be of particular benefit is the mobile core network. The core network comprises a set of services that handle user management and connecting users to each other and the Internet. A diverse set of services such as this with varying user load and high-reliability requirements is a good fit for container orchestration tools such as Kubernetes since it comes with many built-in features to deal with redundancy, auto-scaling services and networking between services.

In this paper, I introduce 4GKube, a new design for orchestrating a 4G mobile core network with Kubernetes that utilises its features to minimise development cost and increase system performance. A partial implementation of this design is presented and made fully open-source to enable further research on this topic. The advantages of taking the approach in this design include auto-scaling of the Mobility Management Entity (MME) with very little work, ease of deployment due to an existing pool of public knowledge, self-healing clusters and access to a large selection of open-source software to help administrate the cluster. I also propose two novel information leakage attacks on Kubernetes and show that they are not feasible in the mobile core network context.

Despite there being many existing efforts to containerise the core network, many of them do not make full use of the potential of containers. For example, [40] places all core network components into a single container, preventing independent scaling and redundancy of components. The project `openair-k8s`[5] gets closer to a useful deployment of the EPC on Kubernetes but ignores some core Kubernetes features such as dynamic application discovery through services and built-in auto-scaling of applications.

The remainder of this paper is structured as follows. Section 2 lays out a detailed back-

ground to both mobile networks and Kubernetes. Section 3 introduces the design of the EPC over Kubernetes. Section 4 follows with an account of the implementation of this design. Section 5 evaluates the design and presents a framework for evaluating an eventual implementation. Section 6 introduces the novel attacks as well as some Kubernetes security advice relevant to deploying a mobile core network. Finally, section 7 concludes.

Chapter 2

Background

2.1 Mobile Networks

Mobile networks are the high-speed, long-range networks that our mobile devices rely on for Internet connectivity and communication. The architecture and specification of these networks are standardised by a body named 3GPP to enable network operators to interact together and equipment manufacturers to build compatible devices. Generally, the architecture for a mobile network consists of two parts; the core network and the radio access network (RAN). This is illustrated in Figure 2.1.

The RAN is composed of user equipment (UEs) and base stations which are named eNodeB in 4G and gNodeB in 5G. The core network has a more complex architecture depicted in Figure 2.2.

The RAN is responsible for managing the wireless portion of the network while the core network is responsible for user management, connecting users to the Internet and each other alongside other administrative functions such as billing.

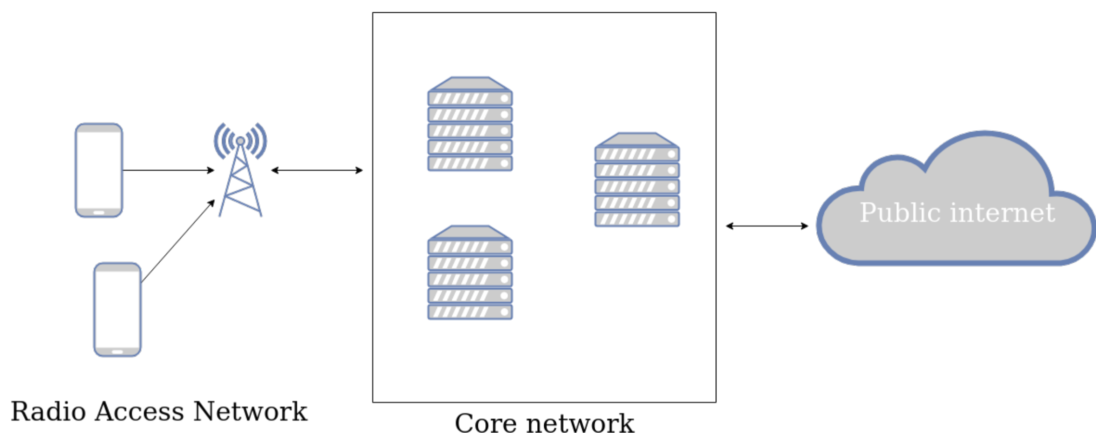

Figure 2.1: Architecture overview of a modern mobile network

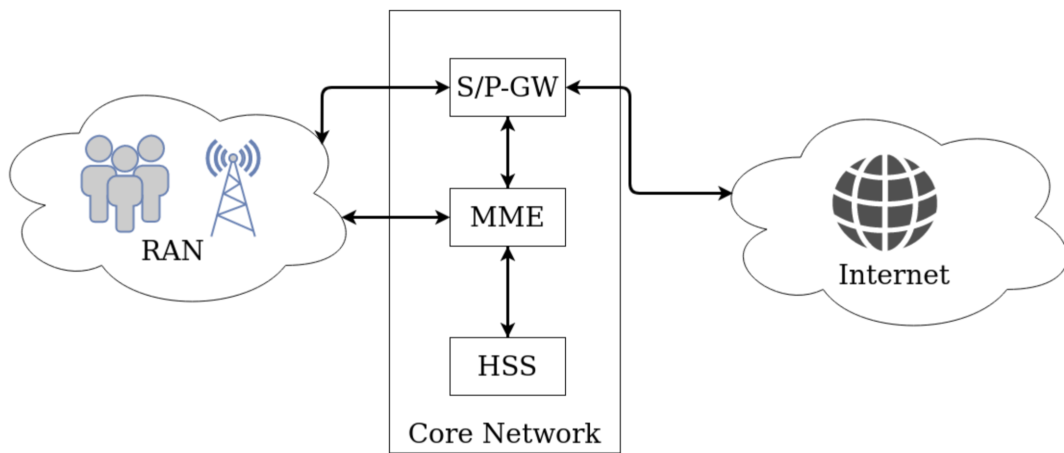

Figure 2.2: Main EPC services and their communication

2.1.1 Evolved Packet Core in 4G

At the time of writing, the most prevalent generation of mobile network in the UK is 4G. 3GPP first specified 4G networks in their Release 8 [1] in 2008. The core network design for 4G networks is called the Evolved Packet Core (EPC), and many mature open source projects [6, 22, 35, 48, 12] have implemented it. I outline the main network functions below, and Figure 2.2 illustrates their communication.

- MME (Mobility Management Entity)

  The MME is responsible for many functions including authentication, management of idle UEs, handling the initial attach of a UE to the network and lawful interception of signalling.

- HSS (Home Subscriber Server)

  The HSS is a database for user and subscription information. It contains some functionality to do with authentication, authorisation and call/session establishment.

- S/P-GW (Serving Gateway and Packet Data Network Gateway)

  In the 3GPP specification, these are two different components, but many implementations, including OpenAirInterface, merge them into one component. Together they handle connection from the mobile network to external networks such as the Internet. 3GPP Release 14, splits these into separate control and user plane components, implemented in OpenAirInterface as the S/PGW-C and S/PGW-U.

While there are other network functions in the EPC architecture, they are not crucial to the running of a mobile network and the open-source implementation I work with for this project, OpenAirInterface (OAI), does not implement them.

2.1.2 4G and 5G

The first 5G specification came in Release 15 [2] in late 2018 and features a new core network design named 5G core (5GC). EPC and 5GC are very different. 5GC uses a service-based mesh architecture, which means that each function exposes a standard API which can be called by any other function. Each function's responsibilities are also now more focused when compared to the larger functions in the EPC architecture. An architecture with many services that each have few responsibilities is called a "micro-service" architecture.

2.1.3 Virtualised and Cloud-native network functions

Traditionally, specialised single-purpose hardware would implement network functions such as the MME. This approach proved to be costly and inflexible as each new network function would require new hardware. In 2012, a white paper from ETSI [13] introduced network function virtualisation (NFV) to solve this problem. NFV moves network function implementations entirely into software, meaning that commodity hardware and virtual machines (VMs) are sufficient to run them. This shares resources more efficiently and enables network architectures to be changed more easily.

More recently, the focus has turned to cloud-native network functions (CNF). These also fall under the umbrella of NFV but are implemented with containers instead of virtual machines. Containers share the underlying operating system, which means they make more efficient use of resources than VMs. They can also be easier to deploy than VMs due to the rise of container image repositories and less underlying software required to make them work.

2.2 Kubernetes

Kubernetes is a container orchestration platform that offers features such as load balancing, self-healing clusters and scalability across many nodes. According to a 2019 report by a cloud monitoring company Datadog [16], Kubernetes is run in 45% of containerised environments, meaning it is the largest player in the container orchestration market.

For organisations looking to move to containerised infrastructure, Kubernetes would be a natural choice. Kubernetes can be used to orchestrate any application, making it applicable to managing network functions as well as more traditional cloud applications such as web servers. Researchers have been studying containerised network functions since at least 2015 [15] and the use of containers could bring many benefits to the implementation of a mobile network core.

2.2.1 Kubernetes concepts

Kubernetes introduces many new concepts, and I have used their names liberally through this dissertation. Some important ones are defined below. Figure 2.3 illustrates how

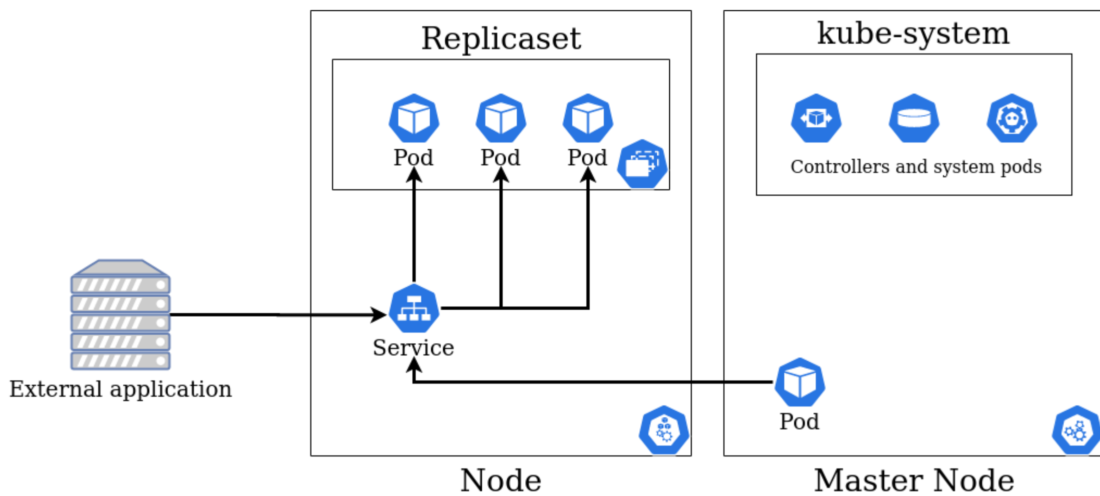

Figure 2.3: How Kubernetes concepts relate to each other

they relate together.

- Clusters

  The cluster is the name for an entire Kubernetes instance including all nodes, pods and system functions.

- Nodes

  A node in a Kubernetes cluster is a physical machine that forms part of the cluster.

- Pods

  Pods are the smallest unit that Kubernetes can schedule (match to a node). They can contain one or more containers as well as the storage mounts for them and network interfaces. A pod corresponds to a virtual or logical host.

- ReplicaSet

  A ReplicaSet is an object that ensures a certain number of pods are running at any given time.

- Deployments

  A deployment specifies an update to the state of some pods or a ReplicaSet. It is the standard way of creating and updating sets of pods.

- Services

  A service in Kubernetes exposes a set of pods under a DNS name. Each pod in Kubernetes can, by default, reach every other pod on the network through unique IP addresses. A service can expose a set of pods and load balance between them through a single DNS name. A service can also expose these pods outside of the cluster.

- Controllers

A controller in Kubernetes has some desired state and attempts to bring the actual state of the world closer to this desired state through a control loop. There are controllers in Kubernetes for everything from ensuring ReplicaSets function to managing storage mounting. It is also possible to write a custom Kubernetes controller.

2.3 Related work

A significant component of this work focuses on scaling the MME using Kubernetes' built-in features. This is due to other works previously identifying the MME as a bottleneck [8, 7, 11, 49]. Accordingly, many of the related works below are to do with scaling the MME. Some of the works are more generally to do with orchestrating the EPC with containers.

In [7], the authors split the MME into micro-services and orchestrate them using Kubernetes. They also create an application-layer load balancer specific to the MME. My work is different as I do not split the MME into microservices; instead, I remove the state from it while leaving it as a monolith. I also use the built-in Kubernetes load balancer reducing development cost.

The authors of [11] propose a system, SCALE, that enables virtualisation and scaling of the MME. Their system does not rely on an orchestrator such as Kubernetes; instead, it manages its pool of MMEs itself. A later system from the same group, ECHO [41], builds on this work by extending the distributed nature of SCALE to the entire core network. That work changes the EPC much more than my work and handles its own distributed challenges instead of relying on off-the-shelf solutions such as Kubernetes.

In [49], the authors implement an MME that scales by using an architecture described in a previous paper [50]. The architecture removes the state from the MME and stores it in a separate database. This database is then used by processing nodes that take the place of the MME. I have similarly scaled the MME; however, this paper is platform agnostic and does not refer to the enabling technologies I have used.

The project `openair-k8s` [5], whilst not published academically, is close in scope to mine. It containerises each of the EPC components and orchestrates them using Kubernetes. The project does not, however, utilise Kubernetes' full potential since it relies on static IP addresses and single instances of each network function. A lone developer, Christopher Agidun, builds a similar system to `openair-k8s`, documented in a blog post [4]. His system uses the Open5GS [35] EPC implementation instead of OpenAirInterface.

There are multiple proprietary implementations of a containerised mobile core network [31, 53]; however, none of them has published their implementations; therefore I am unable to compare my project to them.

Chapter 3

4GKube Design

3.1 Scope

3.1.1 Focus on the Core network

In this dissertation, I will be focusing on the core network. There are a few reasons for this, laid out below.

- The core network is composed of a greater number and variety of services than the RAN, something which fits nicely with a container orchestration model.
- The core network could benefit most from scalability as it is the single point of contact for many base stations. Scalability is a core feature of Kubernetes.
- A RAN requires physical components by definition which do not always containerise well, and so there has been less interest in cloud-native network functions for base-stations. A potential cluster would also have to disregard UEs as the operator does not deploy these.

3.1.2 Reasoning for using EPC

I chose to base this project on the EPC instead of the 5GC for a few reasons. Firstly, the projects that implement the EPC are much more mature and easy to work with than the few projects implementing 5GC. The one project I found implementing release 15 and higher, free5GC [23], had its first standalone 5G release in October. OpenAirInterface has releases dating back to 2016.

Despite the EPC being older than 5GC, deploying it on Kubernetes presents many of the same motivations and challenges that deploying 5GC would. For example, Kubernetes provides auto-scaling capabilities independent of the deployed application.

Another factor in the decision to use the EPC is the current high usage of 4G networks. 5G networks are still very new, and 4G will take a long time to disappear meaning that work on the EPC is still significant. On top of this, the 5G specification provides the

ability to operate alongside existing 4G core networks by only deploying a 5G RAN that connects to them.

3.1.3 Reasoning for using Kubernetes

Kubernetes is not the only container orchestration solution available; however, it is the most popular. Using the most popular option means that this project is more likely to be directly applicable to real-world implementations of cloud-native core networks. There are several direct and indirect competitors to Kubernetes, and I lay out the reasoning for choosing it over some of the most popular ones below.

Docker Swarm [17] is an orchestration tool built into Docker that unites multiple nodes into a single cluster for running containers. Docker Swarm is the tool that is most similar in scope to Kubernetes in this list; however, it a simpler tool and has a less complete feature set than Kubernetes. For example, Docker Swarm does not provide an auto-scaling capability, only a manual scale command. Auto-scaling is a crucial feature for this project, and so Swarm was not chosen.

OpenShift [25] is a container orchestration platform built on top of Kubernetes by Red Hat. It contains all the features of Kubernetes, being based on it, but includes opinionated choices for other tools that come as part of managing a cluster such as image registries, networking infrastructure and logging. `openair-k8s` [5], the project that my implementation is based off, is designed to be deployed on OpenShift. I didn't use OpenShift for this project because it brings few benefits over using vanilla Kubernetes and is a less-open project with a smaller community.

OpenStack [46] is a tool used to manage pools of compute, storage and networking resources meaning that in general, it operates at a lower level than Kubernetes. OpenStack is generally more concerned with managing the underlying hardware than the applications on top of it; OpenStack can even be used to deploy a Kubernetes cluster. OpenStack wasn't chosen as the substrate for this deployment since it is much more low-level and requires more configuration and knowledge than managing a Kubernetes cluster does.

3.2 Challenges

The EPC specification assumes each component to be stateful, including the MME, which poses a problem for the design. Having multiple MMEs behind a layer-4 load balancer could cause different MMEs to serve a UE across two requests. If their state concerning that UE differs, then that could cause problems, potentially forcing the UE to perform an expensive re-attach operation or lose service. Similarly, a scale-down operation from the auto-scaler could permanently delete state from the core network.

The EPC's design does not connect every component like the 5G mesh design. Instead, the EPC has interfaces defined between individual components, each of which is usually on a unique subnetwork. For example, the S6a interface connects the MME to the HSS. The Kubernetes networking model interconnects all pods on the same network so that they can all communicate by default which is contrary to the EPC design. To

resolve this, extra software to allow multiple networks within a cluster will have to be deployed, such as Multus CNI [28].

3.3 Goals

Deploying a mobile core network on Kubernetes only makes sense when using Kubernetes-specific features that are not present or are hard to implement in more traditional deployment methods. Kubernetes has many features, but the most important ones for this design are outlined below as goals.

- Self-healing cluster.

  Kubernetes can recognise if a pod has crashed or if the node it is running on is suffering a failure. In either of these cases, Kubernetes can re-schedule the pod, thereby automatically healing the cluster.

- Auto-scaling.

  Kubernetes can horizontally scale a service by increasing or decreasing the number of instances of it running. It can do this by monitoring the resource usage of each instance and changing the number of instances if this usage passes certain thresholds.

- Function discovery.

  In traditional EPC deployments, the IP address of each network function has to be known ahead of time. Kubernetes obviates this need through Services. Services enable load balancing amongst a group of pods by putting them all behind a single domain name within the cluster.

There are other requirements for this design due to being a core network deployment. Below is a list of requirements ensuring that the mobile network still functions. Any requirements to do with interconnecting to other mobile networks are left out.

- Two users connected to this core network should be able to make a phone call to each other.
- A user connected to this network should be able to send an SMS to another user on this network.
- A user connected to this core network should be able to access the Internet.
- A base station running outside of the cluster should be able to connect itself and its connected user equipment to the core network.

Chapter 5 will assess whether the design meets these requirements as part of the evaluation.

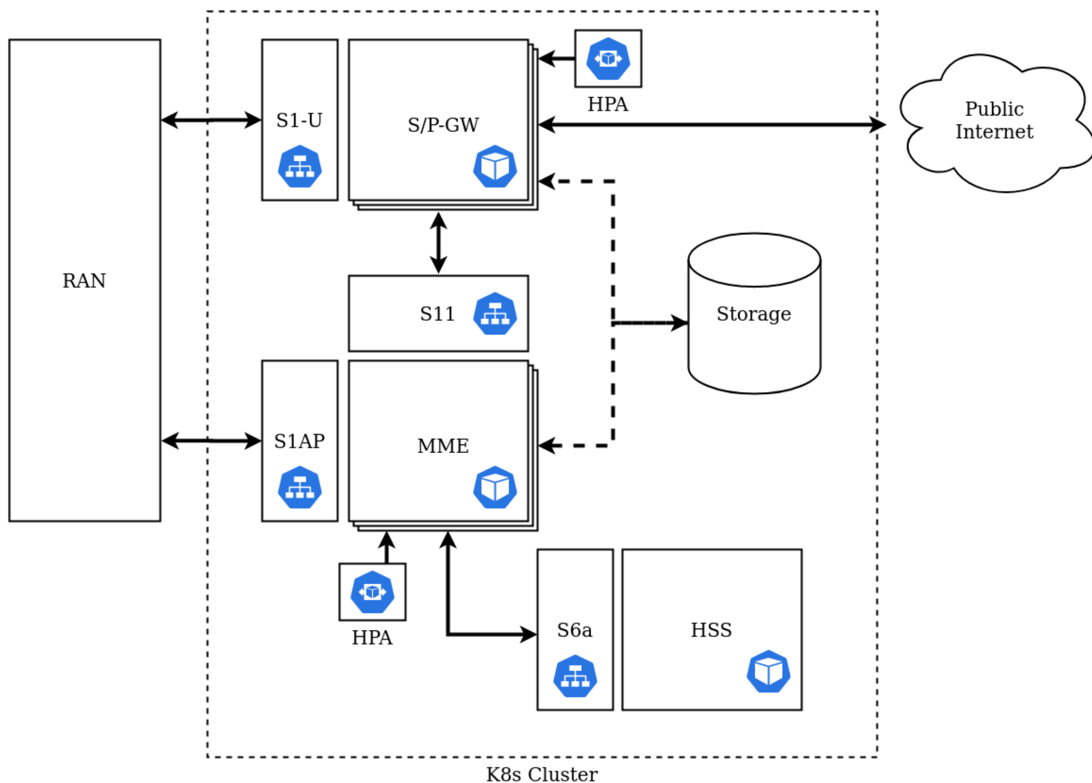

Figure 3.1: Architecture of EPC over Kubernetes system

3.4 Architecture

Figure 3.1 demonstrates the architecture of the system. Examining the diagram reveals a few essential points. Firstly, I mapped the network functions within OpenAirInterface to individual Kubernetes pods. The unit of scaling in Kubernetes is the pod, so if we want individual network functions to be scalable, then there needs to be one function per pod instead of having them all within one.

The double-headed arrows in the diagram represent connections between components. The arrow between the HSS and MME means that these two components communicate; however, the HSS cannot communicate directly with the S/P-GW since there is no arrow connecting them. These connections follow the EPC design, and each operates on a separate subnetwork.

Two MME interfaces are behind a service. These are the interfaces on which the MME services requests. The MME does not have a service for the S6a interface since the MME initiates all requests on this interface. The MME also gains a database, which is where all the state of MME transactions will go. Removing the state enables the MME to become a stateless processing unit as described in 3.5.

The HSS is also behind a service for the S6a interface. This service enables discovery of the HSS without the MME having to hard-code its IP address. On start-up, the MME will attempt to connect to the HSS, which in OAI would happen using its IP address; this service obviates the need for that IP address, replacing it with a DNS name.

The MME is replicated in this design, whereas in traditional deployments of the EPC, it could only have one instance. The MME does the majority of the control-plane work in the EPC, which is why I chose to replicate it. Other works have also identified the MME as a bottleneck and attempted to improve its design [8, 7, 11, 49]. The MME has a horizontal pod autoscaler (HPA) attached for this purpose. Without real usage statistics, it is hard to decide on a suitable metric for the HPA. The metric will most likely be CPU usage, but this could change with the data gained from a real implementation.

The S/P-GW also has a HPA attached and sits behind a service. The S1-U service that the S/P-GW uses will load balance across instances of the component, which means that the S/P-GW needs to become stateless as the MME did. The S/P-GW instances connect to the storage element to store their state. The S/P-GW is replicated in traditional deployments of the EPC as well, but the autoscaling and load-balancing are specific to this implementation.

The interfaces to the RAN will need to be accessible by entities outside of the cluster. External-facing services for the S1AP and S1-U interfaces enable this. A service in Kubernetes becomes external-facing by setting the service type to be one of `NodePort`, `LoadBalancer` or `ExternalName`. The exact type is unimportant to the design and depends on how the network operator has configured the RAN. Kubernetes can expose SCTP services, but some intermediate networks may drop SCTP packets if operating on a public cloud so a tunnel over UDP/TCP may be needed.

3.5 Changes to the MME

The MME is a stateful component in the specification of the EPC, which can cause problems as described in 3.2. Removing the state from the MME to make it into a stateless processing unit solves these problems. The architecture accommodates this through the addition of a database, which stores the state that the MME previously would. In [49], the authors identify the state held by the MME, listed below.

- Authentication information
- Location information
- Routing information for UEs
- UE connection state (Deregistered, CONNECTED, IDLE)
- SCTP connection information

Removing this state from an MME implementation is not a trivial change, but is a necessary component for automatically scaling the MME. In order to scale the MME without Kubernetes, it would still be necessary to make the MME stateless. This change would also have to be followed by creating solutions for monitoring MME load and creating/tearing down instances of the MME. Kubernetes simplifies this process greatly.

3.6 Generalisation to 5GC

Unlike 4G, the 5G core network design takes into account VNFs and CNFs. As a result, the architectural changes make applying the 4GKube design to 5G easier, as seen below.

- Mesh architecture

  The 5GC has a mesh architecture, meaning every service can talk to every other service. In Kubernetes, the network model dictates that every pod should be reachable from every other pod. Services are also discoverable from within the cluster, following this paradigm.

- Micro-services

  The move to micro-services in the 5G architecture means there is an increased number of functions and a decrease in responsibility per function. This change fits into the Kubernetes model well, as it is designed for orchestrating large numbers of functions.

- Stateless functions

  3GPP Release 16 introduces the Unstructured Data Storage Function (UDSF) [3] to the 5GC. This function is a cloud database that stores the state other functions usually would. The UDSF enables other functions to become either stateless or state-efficient, where the function keeps a cache of state but otherwise stores it in the UDSF. Vendors such as Metaswitch and Ericsson [39, 21] are starting to produce stateless implementations of the core network as a result. This move to stateless functions means that changes to network function implementations may not be necessary to orchestrate the 5GC with Kubernetes.

Chapter 4

Implementation

Due to time constraints, a complete implementation of the design in chapter 3 was not possible. However, a port of `openair-k8s` [5] to vanilla Kubernetes and the Powder platform has been accomplished. The following sections go into more detail about exactly what has been accomplished, and the challenges faced when doing so.

4.1 Pre-existing work

Due to the increasing availability of open-source software and open platforms for researchers, there was a large pre-existing body of work to build on.

4.1.1 Powder

Powder [43] is a testbed for wireless networks run by the University of Utah and provided for free to researchers across the world. It provides compute resources as well as physical radio hardware to run experiments.

I chose Powder as the platform for this implementation for two reasons. Firstly, Powder is a mature platform that is easy to work with, providing an online interface to control experiments and a Python API for creating experiment profiles. Secondly, Powder encourages repeatability in research since it instantiates each experiment completely freshly, whether on a real PC or a VM. Powder's profiles also help enable repeatability, as described below.

A core concept in Powder is the idea of a profile. A profile contains experiment configuration including the hardware to be used, the software to install, and the disk images to use. A user can also make their profile public for other researchers to build on. A profile containing a Kubernetes set-up [36] already existed on Powder, and I was able to build on it, as detailed in 4.2.

4.1.2 EPC implementations

Many open-source projects [23, 22, 35, 48, 12] aim to provide a 3GPP-compliant implementation of the mobile core network. At the time of writing, however, the project that is the most mature and has the most industry support is OpenAirInterface [6]. OpenAirInterface (OAI) provides a 3GPP release-10 compliant implementation of the MME, HSS and SPGW components of the core network as well as an implementation of the RAN and a UE simulator.

4.1.3 openair-k8s

One open-source project called `openair-k8s` [5] has already looked at putting OAI onto Kubernetes. Each network function in the core network is containerised and placed into a Kubernetes pod. The eNB also runs in the same cluster as another pod. Container images are built on Red Hat Enterprise Linux (RHEL) systems due to being based on Red Hat-supplied images. The project is designed to be deployed on OpenShift, Red Hat's proprietary container platform that is built on Kubernetes.

While my implementation was based primarily on this project, I had to modify it, which I document in 4.3.

4.2 Creating a Powder profile

4.2.1 Existing profile

A profile to create a Kubernetes cluster [36] existed on Powder which I based my own off. It instantiates an N-node cluster with an example micro-service application deployed on it. Due to the age of the profile, it was using old versions of both Kubernetes and Ubuntu, a problem since in the intervening time Kubernetes had become more mature and added many features. The profile had no other features related to running core networks.

4.2.2 Creating a profile

Creating a profile on Powder is required for running a custom experiment such as the work I was attempting. A profile in Powder is created using a Python file which defines resources through a library called `geni-lib`. Learning `geni-lib` is a challenge as the documentation is only partially complete.

My Powder profile is in a public GitHub repository, a deliberate choice to allow others to build on this work. Putting the profile on GitHub also has the advantage of updating the Powder version with just a `git push` command. I summarise the features that I built on top of the existing profile below.

- Deployment of OAI on Kubernetes.

  OpenAirInterface is a mature implementation of the EPC and the one that I chose

to base my deployment off. Section 4.3 describes the modification and deployment of it.

- Deployment of the Kubernetes metrics-server [27].

  The metrics-server enables resource usage of pods to be monitored, something that is necessary for auto-scaling of pods to occur. It was also pre-requisite for the security experiments carried out in chapter 6.

- Option to run on VMs or real hardware.

  Powder is an environment that is shared with other researchers around the world and has finite resources. While developing the profile, it was important to use resources conservatively to allow other researchers to use the testbed as well, so I added the option to deploy the profile over virtual machines as a toggle.

- Public IPs assigned to nodes.

  Public-facing IPs, combined with Kubernetes services, allows machines outside of the cluster to communicate with pods running inside it. Allowing external access is essential for my design since base stations run outside of the cluster.

- Upgrades to Kubernetes and Ubuntu.

  I upgraded the operating system deployed on the nodes to enable accessing more up-to-date packages for Kubernetes. Upgrading Ubuntu helped to make the experimental environment more representative of the state of the art, which may affect results due to the use of a newer kernel.

- Upgrade Helm.

  Helm is a package manager for Kubernetes and old versions required a service account with administrator privileges to function, which was generally considered bad security practice. I upgraded the Helm version to one which did not require this.

The biggest challenge unrelated to the features above was that the install script on the master node was unable to get the number of nodes in the experiment. The install script needs to know how many nodes are going to join the Kubernetes cluster so that it can exit once they have all joined. The profile I was basing off did this by running a command on the Powder administrator machine, `ops.emulab.net`, but when I ran it, the SSH command failed. The command failed because my Powder SSH key was not on the master node of the experiment since it was a personal key.

The correct way to get parameters from the instantiated profile, such as the number of nodes, is by using a command called `geni-get` as documented in the Cloudlab Manual [37]. Unfortunately, the command does not work as documented on Powder and I had to use a script to replace it [42]. This script gets the parameters through a command on the local machine, negating the need to SSH into another machine.

Figure 4.1 shows the profile options I built during instantiation of the profile.

Figure 4.1: Instantiating the profile in Powder

4.2.3 Kubernetes cluster

Kubernetes does not enforce many specific components on the cluster administrator; instead, it allows the administrator to make their own choices as to how to build their cluster. In this way it acts more like an operating system than an application, meaning that two Kubernetes clusters could be very different just as two installs of Linux could be. Because of this, I had to make some design choices when building the Kubernetes cluster.

The Kubernetes cluster was created with a tool called `kubeadm` [32]. I chose `kubeadm` over alternatives such as `kubespray` [34] and `kops` [33] as it is generally considered simpler and was already being used by the profile I based my own off.

A Kubernetes cluster relies heavily on a virtual networking fabric since each pod can only communicate to the rest of the cluster through a virtual network. The networking fabric I chose for this deployment is called `flannel` [14]. `flannel` is simple to deploy and comes with support for the SCTP protocol built-in, on which the core network relies. I had to modify the installation with an extra `static` plugin from the CNCF's example plugins repository [51]. This plugin was needed because `openair-k8s` relied on assigning static IP addresses.

One issue I ran into when deploying `flannel` was that DNS queries were failing from within pods. This bug was because the subnet `kubeadm` used to instantiate pods was different from the default `flannel` subnet. This bug was hard to spot and was present in the profile I based mine off.

4.3 Deploying OpenAirInterface on Powder

`openair-k8s` is a project to deploy each of the network functions present in OAI as a container in a Kubernetes cluster. It was designed to be hosted in a proprietary Kubernetes-based container management system called OpenShift [25]. Due to the reasons set out in 3.1.3, I chose not to use OpenShift for this project, and so I had to modify `openair-k8s` to run on vanilla Kubernetes.

4.3.1 Building container images

The first step in setting up the project was to build the container images. Since the project has close ties with Red Hat, the base image for each container is a Red Hat Universal Base Image (UBI), that is only accessible through a Red Hat Enterprise Linux (RHEL) subscription. Thankfully, Red Hat provides a no-cost developer subscription for individuals. Using the developer subscription, I installed RHEL on a personal laptop and built the OAI images before uploading them to Docker Hub, a free-to-use image hosting platform.

4.3.2 Configuration changes

Due to `openair-k8s` being a relatively new project and being based on OpenShift, there was a series of configuration changes required to get the project to deploy successfully on my Kubernetes cluster.

- Change the deployment configuration to get container images from my Docker Hub profile instead of the cluster-hosted image repository used in OpenShift.
- Change the SSL certificate creation script to work with vanilla Kubernetes. The MME and HSS were configured to communicate using TLS, but the certificate creation script that came with `openair-k8s` did not work on my set-up. It did not work because OpenShift comes with OpenSSL configuration files built-in, but my Kubernetes environment did not. After creating a configuration file for OpenSSL and modifying the script to use it this worked.
- Change the default network interface to be the one present on Ubuntu. In practical terms, this meant that the default interface on Red Hat's development machines was called `bond0` whereas on my Ubuntu machine it was called `eth0`.

4.3.3 Deployment

Due in part to the intended deployment environment not being vanilla Kubernetes and in part to the immaturity of the project, there was an undocumented requirement for the Multus container network interface (CNI) [28] to run the `openair-k8s` project. Multus is a piece of software that enables multiple network interfaces to be attached to each Kubernetes pod. This feature is necessary for an OAI deployment since each network function has individual interfaces for each connection to another function.

I added a flag to the deploy script that can be enabled or disabled from the Powder profile configuration screen to deploy my version of `openair-k8s`. When enabled, the deploy script will install Multus and then deploy the OAI containers.

After deploying `openair-k8s`, I ran into a problem where the base station, which was running inside the cluster, could not connect to the MME. The reason for this was that the eNodeB was attempting to connect to the MME before it had started, and a temporary fix was to restart the eNodeB pod. This error was not a flaw in the implementation as the eNodeB would eventually be run outside of the cluster and would be started after the core network manually. If this was not the case, this could be fixed with an

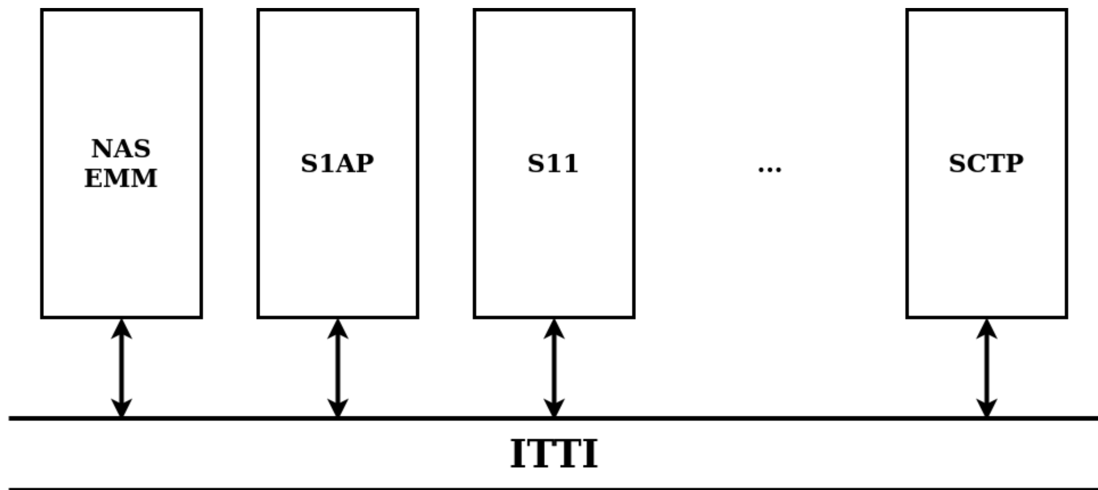

Figure 4.2: Architecture of MME within OpenAirInterface

`InitContainer`, which is a container that runs before other containers in a pod and prevents them from running until it has successfully finished. An `InitContainer` could run before the eNodeB that checks that the MME responds.

Putting the network functions behind services presented a problem since their pods have multiple network interfaces attached. A service in Kubernetes has no concept of network interface and just operates on the default network, meaning that to create a service that uses a secondary network interface requires creating a new Kubernetes object. There is discussion around adding this functionality to Kubernetes [29, 24], however currently this functionality has to be implemented manually, and a few open-source proofs-of-concept exist [26, 54].

4.4 Verifying deployment

Once `openair-k8s` was deployed to Powder, it was necessary to test that it worked properly before changing any OAI code. This deployment would then form a baseline against which to compare code changes. The tool I used to test the deployment was a RAN simulator built by Jon Larrea Martínez of my research group [38].

Unfortunately, I could not get the simulator to connect to the network due to time constraints.

4.5 Modification of MME

As described in the design of 4GKube, the MME state has to be removed and stored in a separate database. The design of OAI helps simplify this change due to the way each network function is architected. The architecture of the MME component in OAI is illustrated in Figure 4.2.

OAI runs a thread for each interface of the MME and for each protocol used. These threads can then send messages to each other on a bus called the Inter Task Interface

(ITTI). Due to this design, removing state can be done methodically by examining each interface, and then each possible message that interface could receive to identify the state that needs to be extracted. Each thread can then have its initial state filled in by a call to the database in the constructor, and subsequent state updates can happen where variables would otherwise be modified.

Unfortunately, I could not complete this part of the implementation due to time constraints.

Chapter 5

Evaluation

5.1 Evaluation of design

5.1.1 Goals

Without an implementation to show the actual performance, I can only evaluate the design by examining how it achieves the goals set out for it in 3.3. The goals of the system are repeated below with the justification of how the design achieves them.

Firstly, the design should make use of the following Kubernetes features:

- Self-healing cluster.

  If a pod crashes in Kubernetes, then the cluster automatically detects this and will restart it. Similarly, if a node experiences a failure or excess stress, then Kubernetes can re-schedule pods on that node elsewhere.

- Auto-scaling.

  The HPA controller implements auto-scaling in Kubernetes. Creating an HPA resource that references a set of pods or a deployment enables auto-scaling for them. The design includes an HPA resource for both the MME and the S/P-GW.

- Function discovery.

  It is impossible to know the IP of a pod in a Kubernetes cluster in advance of its creation; however, Kubernetes has the concept of a "service" that hides pods behind a DNS name. Services, therefore, enable function discovery as the user sets their DNS name in advance. The MME has services for each of its incoming interfaces, which enable discovery of the MME pool.

The other requirements identified for this design were functional requirements to do with the functioning of the network. Since the design of the message flow in the network is the same as in the standard EPC, these requirements will all be met assuming a correct implementation.

5.1.2 Limitations

Even if the design meets the stated goals in theory, there are still some potential issues that may have arisen when implemented. These may include performance, difficulties in implementation or something unforeseen. Alongside these potential limitations, there are more concrete limitations that we can extract from the design itself.

- Lack of scaling for other components.

  The design only includes scaling for the MME and S/P-GW, even though other components may benefit from scaling at higher user numbers. The advantage of using Kubernetes is that it allows for scalability, and it would be nice to make use of that for other components. It is worth noting that it is unknown whether the other components would benefit from scaling, and this would be an avenue for future work.

- Expensive to deploy on existing 4G networks.

  The requirement for the MME and S/P-GW to be stateless means that existing implementations without this feature are incompatible with this design without changes. For operators that have already purchased stateful implementations of the EPC, this design would be expensive to implement.

5.2 Evaluation of implementation

With the implementation unfinished, it is not possible to quantitatively evaluate it. Instead, this section will discuss how I would evaluate the implementation should it have been finished.

5.2.1 Functional tests

Before the performance of the system could be measured, functional tests would have to be carried out to confirm that the system works as expected. These should show that the goals laid out in 3.3 have been met.

5.2.2 Performance tests

The performance of the system would be evaluated under two different conditions. First, the system should be evaluated in a scenario as close to real-world usage as possible by emulating temporal usage patterns that are seen in real networks. Temporal patterns can be derived from data-sets similar to that used in [47], which collected usage data from a tier 1 mobile network in the United States over 32 hours. Secondly, the system should be evaluated against an extreme volume of traffic to test its limits. It would be useful to know what user load the system can handle.

These two scenarios could be tested without the use of a physical RAN by setting up a simulator that works in place of an eNodeB and simulates user traffic. Base station simulators that are capable of simulating multiple individual UEs already exist [12, 38]. Using a simulator would make it feasible to test with large numbers of UEs.

Evaluating the system will require comparing it to a non-containerised reference implementation as well as a containerised baseline without the scaling and load-balancing my design introduces. Some quantitative data that would be useful in this comparison are listed below.

- Time for completion of MME operations. The MME is an important component that has been changed by this design, and so evaluating how long it takes to complete various MME operations compared to a reference system would provide a good measure of its performance.
- Number of concurrent UEs supported. Increasing the number of UEs connected to the network until an attach message fails or a connected UE loses service would reveal this number. I would expect my design to increase this number due to its scaling capabilities.
- Time taken to scale the system. This time could be measured by suddenly increasing the number of UEs/requests on the system so that auto-scaling thresholds were hit and then measuring how long it took for more MMEs to become active and service requests. It would also be useful to measure the average response time before and after the scaling to see if it improves.

Chapter 6

Security

In my original plan for this dissertation, I was going to examine the security aspects of network slicing on the core network, a novel concept introduced in 5G. Network slicing is where multiple isolated networks, potentially run by different operators, could use the same physical infrastructure. This would make deploying a new network cheaper and faster, and help to accommodate the wide variety of services and requirements that 5G promises to serve. In the context of the core network, slicing can be implemented with VNFs and CNFs.

Network slicing could lead to hosting the core network in either a shared private cloud or the public cloud. Both of these scenarios would introduce new attack vectors, and I planned to identify and implement some novel attacks that could be carried out on the core network as a result. The idea for deploying the EPC on Kubernetes came from this motivation since a Kubernetes cluster could feasibly form part of the operator's cloud in this scenario.

A mobile network operator has to pay more attention to security than the average company since it carries some of its customers' most sensitive data. The following chapter lays out some considerations for securing a Kubernetes cluster for use with a core network before introducing the novel attacks I had planned.

6.1 Kubernetes security

Kubernetes is mature software, and some of the biggest technology companies in the world deploy it in production. Despite this, there are still security considerations to be taken into account when deploying a cluster.

When deploying in the public cloud, or an off-site private cloud, an operator needs to take into account standard cloud security issues. Many authors have surveyed this topic [44, 30, 52].

The official Kubernetes documentation has a useful page on securing a cluster [20]. Containers are harder to secure than standard virtual machines as the operating system is shared. Any vulnerability in the kernel can then be exploited from a neighbouring

container. Given the size of data centre servers, this could become a real risk if running in the public cloud, or even sharing a private cloud as may happen with network slicing.

It is essential to have a good review process for changes in the Kubernetes configuration. A user deploys an app on Kubernetes through a JSON or a YAML configuration file. These configuration files can contain malicious options, such as requesting an excessive amount of resources or giving the application administrator privileges on the cluster. The attacks that follow this section all rely on being able to deploy a configuration file to the cluster without admin oversight.

6.2 Attack Background

A control loop is an important concept for cloud orchestration software such as Kubernetes and OpenStack. A control loop has some desired state, the ability to measure actual state and the ability to make changes that bring those two states closer together. Figure 6.1 demonstrates the standard flow of a control loop. First, a control loop will measure the world state and compare it to the desired state. It will then make appropriate changes to affect this state and then wait a set amount of time for these changes to take effect.

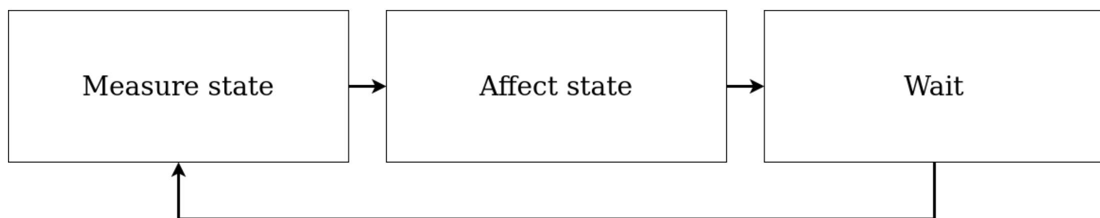

Figure 6.1: Standard control loop structure

A 2018 paper by Baek et al. [10] introduced a side-channel attack on OpenStack that could detect virtual machine creation and deletion events. It did so through affecting the local firewall controller that ran on each physical machine, which used a control loop. Once per iteration of this loop, the malicious VM would make a change to its firewall rules and measure how long it took for that change to take effect. Since the control loop would take a minimum time to execute each iteration, the time it took for the rule change to be applied would be correlated with the load on the controller and therefore with other events happening such as VM creation and deletion.

Kubernetes also uses control loops as a critical component of its implementation. In Kubernetes they are named controllers and are used for many tasks; monitoring pod health, horizontally scaling pods, attaching and detaching storage from pods and more. Users are encouraged to write new controllers if one does not exist for their use case. There is even an officially-maintained template controller [9].

The two controllers I have proposed an attack against below are the replication controller and the horizontal pod autoscaler controller (HPA). The replication controller is responsible for ensuring that the number of pods requested equals the number of

running pods. The HPA is responsible for measuring the load on pods against some set target and scaling the number of pods up or down appropriately.

These attacks assume that load on the cluster is correlated with load on the core network running on the cluster. This correlation would enable a third party that was running on the cluster to glean metrics about the amount of traffic going through the network at any one time, information that is generally kept private by mobile networks. A third party could be running on the cluster either through network sharing; a concept predicted to become more prevalent with the rise of 5G [45], or through gaining access maliciously.

6.3 Attacking the replication controller

6.3.1 Motivation

The replication controller is responsible for making sure the requested number of replicas of each pod are active at any given time. This means that if an attacker can cause their application to crash on purpose, then they create a unit of work for the replication controller, the outcome of which is measurable by their application.

The replication controller runs as part of the `controller-manager` application on the master node. All built-in controllers run as threads inside this application. The replication controller has a control loop period of one second, which means that the execution of each iteration will take at least one second but may take longer.

Detecting how long an operation takes for the replication controller could give insight into the load on the `controller-manager` and master node, thereby giving insight into the load on the cluster. These values should correlate in a scalable, Kubernetes-deployed mobile network to user load since user load would lead to automatic scaling, which would lead to load on the master node and `controller-manager`.

6.3.2 Attack

The attack on this controller consists of two components, the `kill_worker` and the `listening_server` as illustrated in Figure 6.2. The `kill_worker` is responsible for getting restarted and notifying the `listening_server` of the time at which it starts again. The `listening_server` then compiles this information and attempts to derive the load on the replication controller from the recorded cycle times.

The replication controller watches ReplicaSets, such as those created by Deployments, and starts new pods if the number of pods running falls below the requested number. A pod goes through various life-cycle stages but will be restarted by the controller only when considered ”failed”. The ”failed” stage is defined by Kubernetes as follows:

“All containers in the pod have terminated, and at least one container has terminated in failure. That is, the container either exited with non-zero status or the system terminated it.” [18]

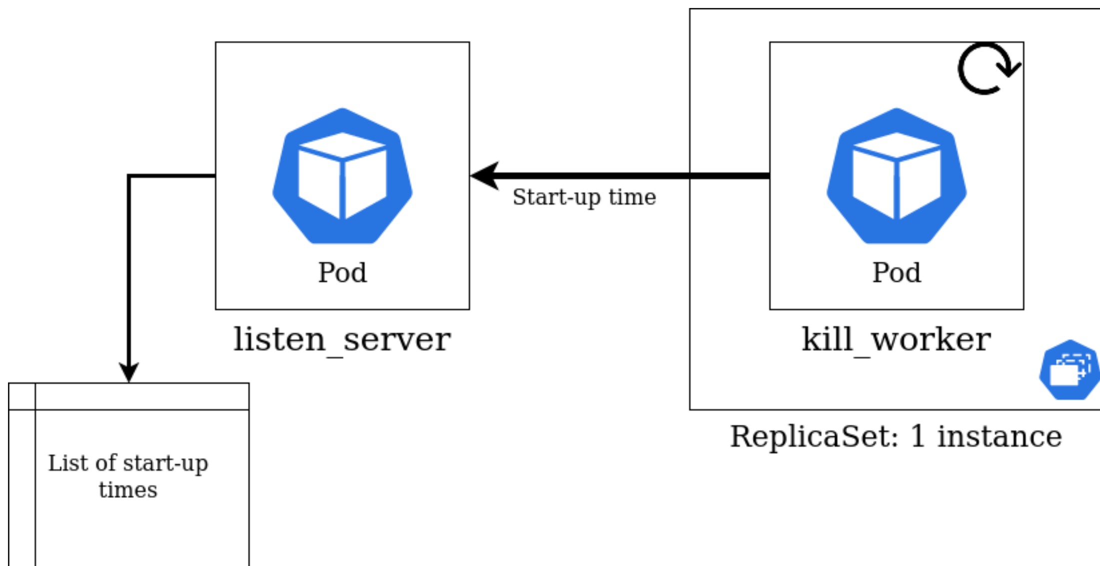

Figure 6.2: Structure of the replication controller attack

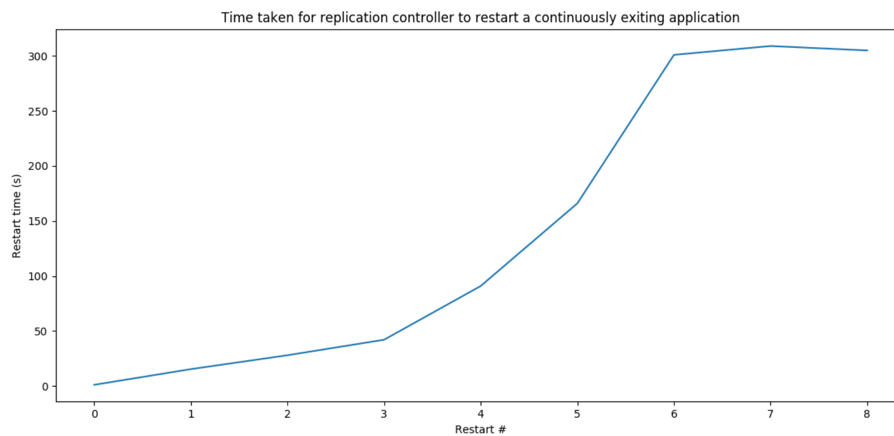

Figure 6.3: Kubernetes restart policy affecting the replication controller restart times

Given this definition, a pod with a single container would be considered ”failed” and restarted if the application in that container exited normally (with an exit code 0). This mechanism is how the `kill_worker` gets restarted; it will send the time it starts up to the `listening_server` and then exit.

6.3.3 Evaluation

The results from running this attack with no other load on the cluster can be seen in Figure 6.3. The graph shows an exponential increase in the time taken to restart `kill_worker`. This is due to the restart policy Kubernetes has [19]. A pod, when considered failed, will be restarted after 10 seconds, then 20 seconds and an exponentially increasing time after that capped at 5 minutes. This back-off resets after 10 minutes of successful execution.

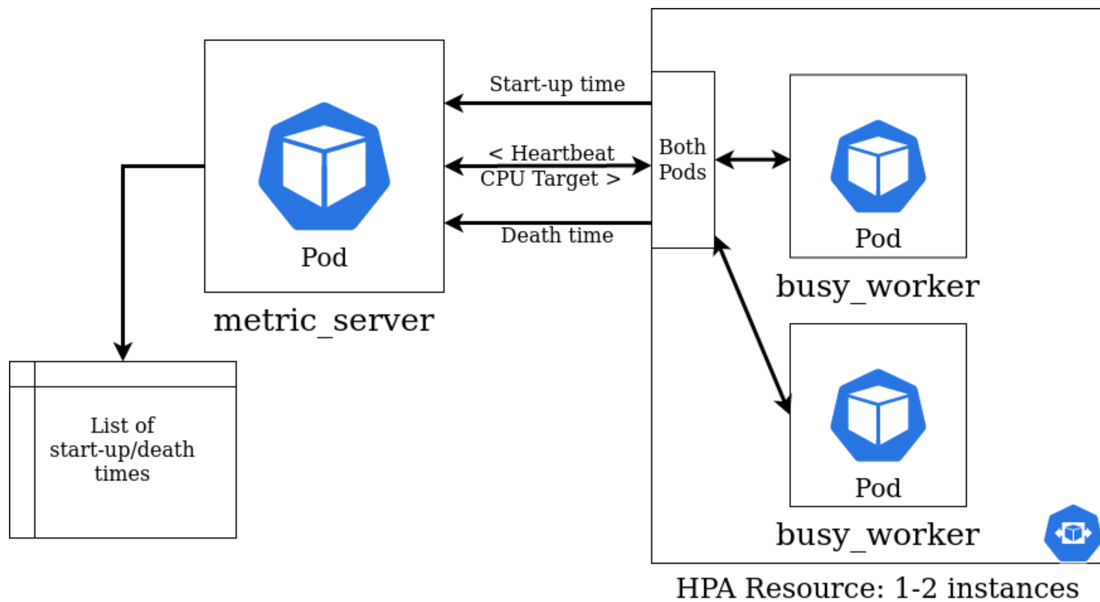

Figure 6.4: Structure of the HPA controller attack system

Unfortunately for this attack, this means that to probe the replication controller with two second resolution, you would need 300 `kill_workers` running (10 minutes / 2 seconds = 600 / 2 = 300). Even 15-second resolution would require 40 instances. Both of these numbers are too large for the size of the cluster that would run a mobile core network, making this attack unfeasible in this context.

6.4 Attacking the HPA controller

6.4.1 Motivation

Similar to the motivation for the replication controller attack, the HPA controller also runs inside `controller-manager`. The HPA controller seems a more promising target since the load on the HPA controller will be the scaling load experienced by the cluster, giving a more direct route to inferring usage statistics.

6.4.2 Attack

The HPA controller checks the average resource usage (CPU, memory or custom) for every group of pods that it monitors before deciding whether or not that group needs to be scaled up or down. This process happens every 15 seconds by default. The HPA controller gets resource usage from a cluster component called `metrics-server` which updates by default every minute.

The attack first requests that the HPA monitors a deployment of `busy_workers`, which are Python servers that can control their CPU usage based on instruction from the `cpu_control` component. The `cpu_control` will then monitor the number of `busy_worker` pods that are active through start-up and death messages. It uses this information to decide whether the `busy_workers` should operate with high CPU usage (cause the HPA

to create more instances) or low CPU usage (cause the HPA to reduce the number of instances). The CPU usage changes every HPA cycle so that the HPA can make a change every cycle. This information is passed to the `busy_workers` through responses to their regular heartbeat messages.

The attack system operates in the stages below.

1. First `busy_worker` pod starts up, sending a start-up message with the time to the `metric_server` and receives an ID in response.
2. The `busy_worker` then sends a heartbeat message and receives a CPU usage target from the `metric_server`. The CPU usage target will be high since there is only one `busy_worker` pod.
3. The `busy_worker` then performs a busy wait for `HEARTBEAT_PERIOD` seconds at the target CPU usage.
4. At the end of this period, the `busy_worker` will send another heartbeat message and get a new CPU target.
5. Eventually the HPA will start a new `busy_worker` pod which will send a start-up message to the `cpu_control`. The `cpu_control` then instructs both pods to have a low CPU usage in response to their heartbeat messages.
6. After the HPA goes round another cycle, it kills the second `busy_worker` due to the low average CPU usage. When it receives the termination signal, it will send a message to the `cpu_control` with the current time before exiting.
7. The cycle repeats.

In this way, the number of pods should oscillate every iteration of the `metrics-server` control loop and the start-up/death messages reveal how long it took for the HPA to make the changes in state. This whole process is illustrated in Figure 6.4.

One caveat of this attack is that in Kubernetes V1.17 it requires an extra flag to be passed to the `controller-manager` on start-up that disables a cool-off period for down-scaling the number of pods. This cool-off period would ordinarily prevent a problem known as thrashing, where pods are continually destroyed and recreated. Thrashing can cause extra work for a real cluster; however, for our attack, this behaviour is desirable. Passing this flag in V1.17 requires administrator access to the master node. In V1.18, however, this is settable on an individual HPA resource basis which means the attack can be carried out by anyone with deploy access to the cluster.

6.4.3 Experiments

6.4.3.1 Preliminary experiment without load

Figure 6.5 shows the results from running the HPA controller experiment with no other load on a two-node cluster. The graph has some interesting features:

- The cycle time is usually measured as around 60s +/- 2 seconds. This time implies that the Kubernetes `metrics-server` component is imposing the limit

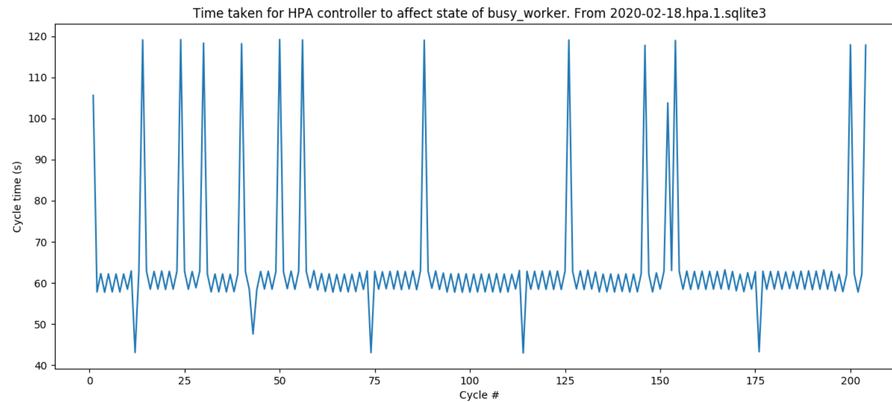

Figure 6.5: Preliminary results of the HPA attack

on granularity since it has a cycle time of 60 seconds.

- The graph is roughly a saw-tooth pattern. Since scale-ups and scale-downs are interleaved, this suggests that scaling up the number of pods takes longer than scaling down the number of pods. Scale-ups take longer because the pod start-up time includes creating a new container, starting Python and running the script to the point it records the time. This extra time could be measured and accounted for in a final implementation.
- There are spikes in the graph that are roughly double the average cycle time. The `metrics-server` recording the resource usage before the `cpu_control` has had a chance to change it causes this. That would lead the HPA to act when it receives updated information from the `metrics-server` a full cycle later.

6.4.3.2 Experiments with load

`metrics-server` resolution (s)	Downscale period (mins)	Avg cycletime (s)	Correlation coefficient
15	30	35.6	0.29
60	90	61.6	-0.30

Table 6.1: Results from HPA experiment with load. Correlation coefficient calculated on mean cycle times of each number of systems running in parallel.

After running the experiment with no additional load, I ran two experiments that ran 10 of the `cpu_control`/`busy_worker` system in parallel to place load on the HPA controller. One experiment ran with a `metrics-server` resolution of 15 seconds and the other with a resolution of 60 seconds. The experiments then progressively killed the instances of the system until just one remained to see the effect of varying the load. The cycle times recorded by the one surviving instance are summarised in table 6.1 and Figure 6.6.

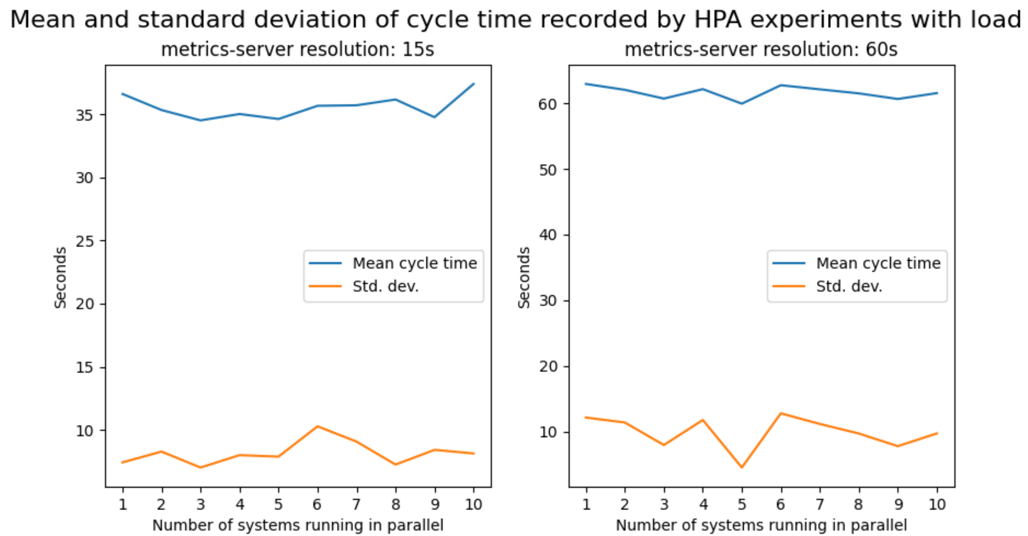

Figure 6.6: Results of HPA experiment with load

6.4.4 Evaluation

The results of the experiments run with the HPA attack show that it is unlikely to work in the context of a mobile core network. If the load on the HPA controller was correlated with cycle time recorded by the `cpu_control` component, then we would expect to see a strong correlation coefficient; however, table 6.1 shows that for both resolutions tried, there is only a weak correlation between the cycle time and the load. For a resolution of 60 seconds, the correlation is even negative, which suggests that factors other than load are affecting the cycle time much more strongly.

It is possible that if a significantly higher number of systems were running in parallel and putting load on the HPA controller, then we would see a correlation between cycle time and amount of load. However, more parallel systems would not reflect the amount of load that a typical mobile core network would generate. Scaling events in a mobile network would be infrequent, only happening when user load significantly grew or dropped, potentially a few times a day. The load in the above experiments was between one and ten requests every 15 seconds, which may already be greater than we would see from a mobile core.

Chapter 7

Conclusion

7.1 Conclusion

In this report, I introduced a new design for orchestrating the EPC with Kubernetes and partially implemented it. I also introduced two potential side-channel attacks on Kubernetes and demonstrated that they would not work in the context of a mobile core network. Through this research, I hope to have demonstrated that Kubernetes and cloud-native network functions are an attractive avenue of exploration for mobile network operators when upgrading their networks.

7.2 Future work

This report leaves many potential avenues for future work, with a few listed below.

- Full implementation of the design outlined in chapter 3. Once a prototype was implemented, it would be interesting to compare its performance with the current implementation as well as a non-containerised version.
- Examining the effects of allowing other components to scale. In the design for 4GKube, only the MME can automatically scale, but scaling other components may also bring benefits.
- An equivalent work for the 5G core. The EPC will lose its relevance much faster than the 5G core, and fledgeling projects are providing an open-source implementation of the 5G core, making this feasible.
- Increasing the scale of experiments for the attacks in chapter 6. The attacks were deemed unfeasible in part due to the scale of a core network. The attacks may work well in a larger scale context, potentially outside of telecoms.
- Examining other orchestration tools. Kubernetes is not the only container orchestration platform available. Others include OpenStack [46] and Docker Swarm [17] and may bring unique benefits of their own.

Bibliography

[1] 3GPP. 3GPP Release 8. Technical report, 3rd Generation Partnership Project (3GPP), 2008.

[2] 3GPP. 3GPP Release 15. Technical report, 3rd Generation Partnership Project (3GPP), 2018.

[3] 3GPP. Unstructured data storage services. Technical Specification (TS) 29.598, 3rd Generation Partnership Project (3GPP), 3 2020. Version 16.0.0.

[4] Christopher Agidun. Virtual 4g simulation using kubernetes and gns3. `https://dev.to/infinitydon/virtual-4g-simulation-using-kubernetes-and-gns3-3b7k`, 2020.

[5] OpenAirInterface Software Alliance. openair-k8s. `https://github.com/OPENAIRINTERFACE/openair-k8s`, 2020.

[6] OpenAirInterface Software Alliance. Openairinterface — 5g software alliance for democratising wireless innovation. `https://www.openairinterface.org/`, 2020.

[7] P. C. Amogh, G. Veeramachaneni, A. K. Rangisetti, B. R. Tamma, and A. A. Franklin. A cloud native solution for dynamic auto scaling of mme in lte. In *2017 IEEE 28th Annual International Symposium on Personal, Indoor, and Mobile Radio Communications (PIMRC)*, pages 1–7, 2017.

[8] X. An, F. Pianese, I. Widjaja, and U. G. Acer. dmme: Virtualizing lte mobility management. In *2011 IEEE 36th Conference on Local Computer Networks*, pages 528–536, 2011.

[9] The Kubernetes Authors. kubernetes/sample-controller. `https://github.com/kubernetes/sample-controller`, 2020.

[10] Hyunwook Baek, Eric Eide, Robert Ricci, and Jacobus Van der Merwe. I heard it through the firewall: Exploiting cloud management services as an information leakage channel. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 361–373, New York, NY, USA, 2018. Association for Computing Machinery.

[11] Arijit Banerjee, Rajesh Mahindra, Karthik Sundaresan, Sneha Kasera, Kobus Van der Merwe, and Sampath Rangarajan. Scaling the lte control-plane for future mobile access. In *Proceedings of the 11th ACM Conference on Emerging*

Networking Experiments and Technologies, CoNEXT '15, New York, NY, USA, 2015. Association for Computing Machinery.

[12] IIT Bombay. Nfv_lte_epc: A virtualized evolved packet core for lte networks. https://github.com/networkedsystemsIITB/NFV_LTE_EPC, 2020.

[13] M Chiosi, D Clarke, P Willis, et al. Network function virtualization: An introduction, benefits, enablers, challenges & call for action. In *SDN and OpenFlow World Congress*, 2012.

[14] CoreOS. coreos/flannel. https://github.com/coreos/flannel, 2020.

[15] R. Cziva, S. Jouet, and D. P. Pezaros. Gnfc: Towards network function cloudification. In *2015 IEEE Conference on Network Function Virtualization and Software Defined Network (NFV-SDN)*, pages 142–148, Nov 2015.

[16] Datadog. 8 facts about real-world container use. https://www.datadoghq.com/container-report/, Nov 2019.

[17] Docker Documentation. Swarm mode overview. https://docs.docker.com/engine/swarm.

[18] Kubernetes Documentation. Pod lifecycle. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle.

[19] Kubernetes Documentation. Pod lifecycle. https://v1-17.docs.kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/#restart-policy.

[20] Kubernetes Documentation. Securing a cluster. https://kubernetes.io/docs/tasks/administer-cluster/securing-a-cluster/.

[21] Ericsson. 5g core (5gc) network. https://www.ericsson.com/en/digital-services/offerings/core-network/5g-core.

[22] Facebook. magma — platform for building access networks and modular network services. https://github.com/facebookincubator/magma, 2020.

[23] The free5GC Authors. free5gc — open-source 5gc. https://www.free5gc.org, 2020.

[24] Kubernetes Network Plumbing Working Group. Npwg service abstraction discussion board - google docs. https://docs.google.com/document/d/1tYs_O7Dz-YQenwPz6QHwm4ZoQ3bu0-1m-2c_7dno4N8/edit?usp=sharing.

[25] Red Hat. Red hat openshift container platform. https://www.openshift.com.

[26] Tomofumi Hayashi. s1061123/multus-proxy-k. https://github.com/s1061123/multus-proxy-k.

[27] Kubernetes SIG Instrumentation. kubernetes-sigs/metrics-server: Cluster-wide aggregator of resource usage data. https://github.com/kubernetes-sigs/metrics-server, 2020.

[28] Intel. intel/multus-cni. https://github.com/intel/multus-cni, 2020.

[29] intel/multus cni. Get multus network ip discovered via k8s service · issue #256. https://github.com/intel/multus-cni/issues/256.

[30] Minhaj Ahmad Khan. A survey of security issues for cloud computing. *Journal of Network and Computer Applications*, 71:11 – 29, 2016.

[31] Heather Kirksey, Azhar Sayeed, and Fu Qiao. Keynote: E2e 5g cloud native network. https://www.youtube.com/watch?v=IL4nxbmUIX8.

[32] Kubernetes. Bootstrapping clusters with kubeadm. https://kubernetes.io/docs/setup/production-environment/tools/kubeadm/.

[33] Kubernetes. Installing kubernetes with kops. https://kubernetes.io/docs/setup/production-environment/tools/kops/.

[34] Kubernetes. Installing kubernetes with kubespray. https://kubernetes.io/docs/setup/production-environment/tools/kubespray/.

[35] Sukchan Lee. open5gs. https://github.com/open5gs/open5gs, 2020.

[36] Richard Li (licai). kube-profile. https://gitlab.flux.utah.edu/licai/emulab-profile/-/tree/3163940d/private-profiles/kubernetes, 2020.

[37] The Cloudlab Manual. 10 advanced topics. https://docs.cloudlab.us/advanced-topics.html.

[38] Jon Larrea Martínez. Ran simulator. Unpublished work, 2020.

[39] Metaswitch. 5g core solution. https://www.metaswitch.com/solutions/mobile-solutions/5g-core.

[40] moffzilla. Oai-docker. https://github.com/moffzilla/OAI-Docker, 2020.

[41] Binh Nguyen, Tian Zhang, Bozidar Radunovic, Ryan Stutsman, Thomas Karagiannis, Jakub Kocur, and Jacobus Van der Merwe. Echo: A reliable distributed cellular core network for hyper-scale public clouds. In *Mobicom 2018*. ACM, October 2018.

[42] Felicián Németh. geni-get-param. https://netsoft.tmit.bme.hu/nemethf/cloudlab-kubernetes/blob/7afc77d0eec5eca12db760c998f89dc0ba5c3bdf/geni-get-param.

[43] The University of Utah. Powder: Platform for open wireless data-driven experimental research. https://www.powderwireless.net, 2020.

[44] Gururaj Ramachandra, Mohsin Iftikhar, and Farrukh Aslam Khan. A comprehensive survey on security in cloud computing. *Procedia Computer Science*, 110:465 – 472, 2017. 14th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2017) / 12th International Conference on Future Networks and Communications (FNC 2017) / Affiliated Workshops.

[45] K. Samdanis, X. Costa-Perez, and V. Sciancalepore. From network sharing to multi-tenancy: The 5g network slice broker. *IEEE Communications Magazine*, 54(7):32–39, 2016.

[46] Omar Sefraoui, Mohammed Aissaoui, and Mohsine Eleuldj. Openstack: toward an open-source solution for cloud computing. *International Journal of Computer Applications*, 55(3):38–42, 2012.

[47] M. Z. Shafiq, L. Ji, A. X. Liu, J. Pang, and J. Wang. Geospatial and temporal dynamics of application usage in cellular data networks. *IEEE Transactions on Mobile Computing*, 14(7):1369–1381, 2015.

[48] Software Radio Systems. srslte — open source sdr lte software suite from software radio systems (srs). `https://github.com/srsLTE/srsLTE`, 2020.

[49] Y. Takano, A. Khan, M. Tamura, S. Iwashina, and T. Shimizu. Virtualization-based scaling methods for stateful cellular network nodes using elastic core architecture. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science*, pages 204–209, 2014.

[50] Motoshi Tamura, Tetsuya Nakamura, Takahiro Yamazaki, and Yuki Moritani. A study to achieve high reliability and availability on core networks with network virtualization. *NTT Docomo Tech. J*, 15(1):42–50, 2013.

[51] CNCF CNI Team. containernetworking/plugins. `https://github.com/containernetworking/plugins`, 2020.

[52] Luis M. Vaquero, Luis Rodero-Merino, and Daniel Morán. Locking the sky: a survey on iaas cloud security. *Computing*, 91(1):93–118, Nov 2010.

[53] Verizon. Verizon and ericsson first in the world to introduce cloud-native technology in a live wireless core network environment. `https://www.verizon.com/about/news/verizon-and-ericsson-introduce-cloud-native-technology`, 2019.

[54] Peter White. k8snetworkplumbingwg/net-service-controller: Network service controller (multus compatible). `https://github.com/k8snetworkplumbingwg/net-service-controller`.