Network Functions Virtualization and Security
Offering campus security by leveraging cloud-native infrastructure

by
Roderik W.G. Janssen, BSc
Rik.Janssen@os3.nl

March 11, 2019

6 ECTS
January - March, 2019

Supervisor:
Dr. Richa Malhotra (SURFnet)

Assessor:
Prof. dr. ir. Cees T.A.M. de Laat

University of Amsterdam
Graduate School of Informatics
Abstract—In this paper we investigated the feasibility of using Network Functions Virtualization (NFV) via cloud-native infrastructure in order to provide SURFnet-based security-related network functions to the campus networks of the Dutch educational institutions. We opted to use cloud-native infrastructure as its definition showed overlaps with the objectives of NFV. We built a proof of concept on top of the Kubernetes scheduler, and further leveraged technologies such as Kata Containers, the NEMU hypervisor, and the Multus Container Network Interface. Additionally, we built a highly available and scalable NFV Controller, as to verify the possibilities and limitations of utilizing cloud-native infrastructure. We concluded that, as per the NFV objectives defined by the European Telecommunications Standards Institute (ETSI), cloud-native infrastructure has a lot of potential, but lacks maturity. We need further integration of existing and upcoming technologies, as to provide a turn-key solution.

Index Terms—NFV, security, Kubernetes, cloud native, virtualization.

1 INTRODUCTION

The security threat landscape is ever changing with cyber attacks becoming increasingly sophisticated and targeted. The traditional approach of securing an educational organization’s network with a firewall is probably not sufficient anymore. In parallel to this development, new network architectures and approaches such as Network Functions Virtualization (NFV) and Software-Defined Networking (SDN) are challenging the paradigms of network design. They claim to be cost-effective, and offer more flexibility, scalability, and security.

As the Internet Service Provider (ISP) and cloud provider of the Dutch National Research and Education Network (NREN), SURFnet is interested in the role it can play in offering security-related network functions to the campus networks of the educational institutions in the Netherlands, and what the usage of NFV will entail in order to achieve this.

The aim of this paper is to investigate the feasibility of using NFV in order to provide ISP-based security-related network functions to campus networks.

1.1 Outline

This paper is divided into eight major sections. Sections 2, 3, and 4 respectively describe the background, research question, and used methodology. Sections 5 and 6 describe and discuss the results. Section 7 looks at the related work, and Section 8 concludes the research. The last section lists possible future work. Finally, there are several appendices listed.

2 BACKGROUND

In this section we will look at the background of NFV and SURFnet, as to clarify the context in which the research has been conducted.

2.1 NFV

NFV is created and defined by the European Telecommunications Standards Institute (ETSI). In [1] and [2] ETSI outlines several objectives and benefits of NFV. The following is a small summary:

- Using Commercial-Off-The-Shelf (COTS) hardware (e.g., generic x86_64 servers) to provide VNFs through software virtualization. This will improve capital efficiencies when compared to dedicated hardware implementations.
- Decoupling functionality from physical location by improved flexibility in assigning VNFS to hardware resources (i.e., assign NFVs to data centers, customers’ premises, etc.).
- "The possibility of running production, test and reference facilities on the same infrastructure provides much more efficient test and integration, reducing development costs and time to market.”
- "Reduced power usage achieved by migrating workloads and powering down unused hardware.”
- Preventing vendor lock-ins by using standardized and open interfaces between the VNFS, the infrastructure, and the management components.
- "Improved operational efficiency by taking advantage of the higher uniformity of the physical network platform and its homogeneity to other support platforms.”

Figure 1 gives a high-level overview of the NFV framework. The framework shows network functions (e.g., routers and firewalls) being implemented as software-only entities, called Virtual Network Functions (VNFS), on top of an NFV Infrastructure (NFVI), as described by ETSI in [1]. The NFVI consists of physical hardware resources, and the virtualization thereof to provide the platform on which the VNFS can be executed.

![Figure 1. High-level NFV framework. Source: [1]](image-url)}
Regarding the connectivity between the VNFs, in [1] ETSI states two types of relations can exist between VNFs: a VNF Forwarding Graph (VNF-FG) or a VNF Set. The latter means the connectivity is not specified (i.e., the VNFs are standalone entities), whereas with VNF-FG the connectivity is specified. In [4] ETSI further specifies a VNF-FG as a "graph of logical links connecting [network function] nodes [where at least one node is a VNF] for the purpose of describing traffic [flows] between these network functions." I.e., a VNF-FG describes the sequence of VNFs that packets traverse. Furthermore, in [1] ETSI states that "[a]n end-to-end network service can be described by an NF Forwarding Graph of interconnected Network Functions (NFs) and end points." Figure A1 in Appendix A shows an example of a VNF-FG. In the figure we can observe the traffic’s path, as it moves through the physical and virtual network functions.

Finally, in [1] ETSI also recognizes the paradigm shift that NFV creates, and the need for future study of multiple areas, including the security aspects.

2.2 SURFnet

Around 2017 SURFnet was informally approached by several Dutch educational institutions (mainly polytechnics), regarding the possibilities of offering security-related network functions. Firewalls, and their configuration complexity as well as financial expenses, were specifically mentioned as motivation. In January of 2018, work done in cooperation with SURFnet, [5], looked at the challenges and opportunities that NFV could provide a campus network, but did not, however, specifically considered the security implications of NFV.

In September of 2018 a pilot project, [6], was started by SURFnet, offering a Firewall as a Service (FaaS). This FaaS platform was built using the OpenStack cloud platform and Juniper’s Contrail Cloud platform (version 4.1). The FaaS platform is a form of Network Functions Virtualization as a Service (NFVaaS); SURFnet provides and manages the VNFs, but the underlying infrastructure remains only accessible to SURFnet itself (i.e., the educational institutions don’t have access to the Contrail management interface).

As a result, new insight was gained, and several new topics of research were started by SURFnet. One of these topics was the further role that NFV could play in offering security-related network functions. Figure 2 gives an overview as to what SURFnet’s goal is with regard to NFV.

As a result, new insight was gained, and several new topics of research were started by SURFnet. One of these topics was the further role that NFV could play in offering security-related network functions. Figure 2 gives an overview as to what SURFnet’s goal is with regard to NFV.

Furthermore, SURFnet indicated the components of its network infrastructure are primarily selected based on Service-level Agreements (SLAs) with vendors. SURFnet itself does the management and orchestration on top of these components.1

3 Research Question

The research question is as follows: "How would ISP-provided NFV contribute to the security-related network functions of campus networks?" The sub-questions below have been defined in order to systematically answer the research question.

1. Which security-related network functions are present within the campus networks of the education institutions in the Netherlands?
2. Which security-related network functions can be provided by NFV technologies?
3. What are the security implications inherent to the NFV technologies/platforms themselves?
4. What are the insights gained, when facilitating security-related network functions of the aforementioned campus networks, when using NFV technologies?

To further clarify sub-questions 3. and 4., the research will focus on both the security aspects of the NFV technologies themselves (i.e., a technical evaluation), as well as the security implications regarding the usage of NFV technologies (i.e., a scenario/use case evaluation). The next section further outlines the approach that was taken to answer these question.

4 Methodology

We focused on the NFV aspects of offering the security-related network functions, the research did not entail a requirement analysis of the educational institutions’ outsourcing policies. Furthermore, we only looked at network functions; higher-level applications (e.g., web servers and database servers) were considered to be out of scope in relation to implementing them as VNFs. The research was technical...
in nature. Furthermore, the research focused on the VNFs themselves, and not on the technologies needed for the steering of traffic to, from, and between the VNFs. Additionally, we define ‘security’ as to mean the confidentiality, integrity (i.e., the authenticity), and availability of (digital) information. This is known as the CIA/AIC triad security model, as stated in [7].

When we look at campus network functions in educational organizations, one would have to account for discrepancies in terminology, and possible overlaps, when cataloging every single security-related network function. Moreover, the response rate of, and offsets between types of interviewees (e.g., large or small organizations) could further obfuscate the results. As the time constraints of this research did not allow for such an assumed extensive requirement/gap analysis, a literature study was conducted to answer the first sub-question instead.

Additionally, the literature from ETSI was again used to answer the second and third sub-question. Furthermore, regarding the third sub-question, we looked at two Request for Comments (RFCs).

Finally, in order to answer the last sub-question, we first set out to combine cloud-native technologies with the Tungsten Fabric platform. We opted for this approach after our results showed an overlap between ETSI’s NFV objectives and cloud-native technologies. To elaborate further, Tungsten Fabric is the successor to Juniper’s Contrail platform, which SURFnet utilizes within their pilot project, and integrates with cloud-native technology.

However, based on our result with Tungsten Fabric, we took a different approach by looking at other cloud-native technologies, and subsequently leveraged these technologies to create a Proof of Concept (PoC) showcasing the feasibility of secure VNFs. We opted for remaining with cloud-native technology, since we already observed the aforementioned overlaps between the definitions regarding ETSI’s NFV objectives and CNCF’s cloud-native technology.

Last, we created a test application in order to further observe the capabilities and challenges of providing the VNFs.

5 Results

5.1 Security-related network functions in the campus

We first limited our inquiry to the previous work done at SURFnet, described in [5], and used its inventory of network functions to select from. We were unable to find additional academic work that listed any security-related network functions (beyond the ones we already described), therefore we looked at other security-related literature to supplement our inventory.

We identified additional functions by looking at several NFV use cases (namely #8 and #15) described by ETSI in [8]. Moreover, when we looked at the work done in the area of cyber security threats in the Netherlands, we found three documents of interest, namely [9]–[11]. Although these documents highlight current trends and vulnerabilities, they lacked in offering specific technical means to be used as mitigation (e.g., which network functions), and focused more on governance techniques instead. Finally, we looked at Cisco Systems’ security reference architecture called SAFE. This architecture encompasses multiple network locations, including the campus, which is specified in [12], and accounts for the architecture, design, and operational domains of business networks.

The literature mentioned above resulted in the following selection of security-related network functions we might find in campus networks:

1. Firewall
2. Virtual Private Network (VPN)
3. Load balancer
4. Proxy / Application gateway
5. Intrusion Detection System (IDS)
6. Intrusion Prevention System (IPS)
7. IPsec/SSL offloader
8. Traffic monitor
9. Key management and key operations offloader
10. Network probe
11. Event generator
12. Honeypot
13. SPAM filter
14. Anti-virus software
15. Domain Name System (DNS) security
16. Identity-based access (i.e., authentication and authorization)
17. Threat intelligence (i.e., zero-day malware and attacks)
18. Web reputation/filtering (i.e., tracking against URL-based threats)
19. Client-based security (i.e., security software, such as a firewall and anti-virus, on client devices)
20. Posture assessment (i.e., client endpoint compliance verification and authorization)
21. Wireless intrusion prevention
22. Wireless rogue detection

5.2 Security-related network functions through NFV

Aside from the wireless and client-based security functions, all network functions seem to be eligible for virtualization when we correlated them to the use cases described by ETSI in [8]. The wireless network functions are assumed to be tightly coupled to the hardware, and client-based security functions are assumed to be outside of the management domain of the NFV provider. Furthermore, other security constrains might also apply. The next section will discuss these constrains.

Moreover, performance penalties might apply, depending on which virtualization technologies are being used. In [13] ETSI mentions the use of technologies like Single Root I/O Virtualization (SR-IOV) and the Data Plane Development Kit (DPDK) as possible performance acceleration techniques. As described in [14], SR-IOV is an extension to the PCI Express (PCIe) specification, and allows a network card to segment its resources among Virtual Machines (VMs), bypassing the virtualization layer. Furthermore, as stated on [15], DPDK is a set of libraries to "accelerate packet processing workloads running on a wide variety of CPU architectures". DPDK does this by bypassing the kernel and implementing the network functionality in user space.

5.3 Existing NFV security work done by ETSI

When looking at the work done by ETSI, we observed that the security aspects of NFV have already been thoroughly
mapped out. The following ETSI publications are related to the security of NFV:

- 'NFV Security; Problem Statement', described in [16];
- 'Security and Trust Guidance', described in [17];
- 'System architecture specification for execution of sensitive NFV components', described in [18];
- 'Security Management and Monitoring specification', described in [19];
- 'Report on the application of Different Virtualisation Technologies in the NFV Framework', described in [20], and 'Use Cases', described in [8].

In the documents listed above we found security technologies, policies, processes, practices, impact analyses, design considerations, requirements, and so on. However, several items stood out. In [16] ETSI specifies several general concerns, one of which is 'safety vs. complexity'.

ETSI observed that the virtualization of networks is difficult for humans to model in their minds. Therefore, virtualization increases the complexity of the overall infrastructure, which can lead to errors being made. Consequently, these error might result in service outages, or leave vulnerabilities that can be exploited by malicious entities.

Additionally, in [17] ETSI observed that the "NFV security considerations are very similar to hypervisor-based virtualisation security considerations in their architecture and interfaces." Hence, ETSI encourages security architects and operations managers to consider use cases beyond mere hypervisor-based virtualisation, as to identify new security considerations. Furthermore, ETSI recognized three major NFV security use cases: Intra-VNFSec, Infra-VNFSec, and Extra-VNFSec. Intra-VNFSec entails the security within VNFs (e.g., sensitive data stored within a VNF), Infra-VNFSec entails the security between VNFs (e.g., secure traffic flows), and Extra-VNFSec entails the security external to VNFs (e.g., the regulatory and jurisdictional impact on VNFs, such as accounting).

The last item that stood out was the usage of OS containers, and its impact on the NFVI. In [20] ETSI observed that because of the nature of operating-system-level virtualization, VNFs that require specific kernel functionality would be hindered. However, ETSI also observed that this could be taken into account during the placement of VNFs, albeit at the cost of introducing more complexity. Moreover, ETSI stated that containers have a larger attack surface, whereas hypervisor-based solutions are more flexible given their support for backwards compatibility with specific kernels. Furthermore, ETSI recognized two areas of concerns when using Linux containers: host kernel access and hardware pass-through. In the case of the former, ETSI stated that a container could compromise the host system when given access to the host kernel. In case of the latter, ETSI observed that isolation techniques regarding resource starvation are immature. However, this security assessment was performed during December, 2015. We will come back to this in Section 5.6.

ETSIs usage of the word ‘safety’ is somewhat ambiguous. Rather, the word ‘security’ might have been a better choice here, considering ETSIs observation regarding the exploitation of vulnerabilities.

5.4 Existing NFV security work listed in RFCs

We found the following two RFCs that are related to network security functions:

- 'Interface to Network Security Functions (I2NSF): Problem Statement and Use Cases', described in [21],
- and 'Framework for Interface to Network Security Functions', described in [22].

The RFCs outline the need for a standard interface to manage network security functions, considering the growing demand for hosted / cloud-based security services, and the heterogeneous nature of existing infrastructures (i.e., the infrastructures consist of on-premise functions, off-premise functions, and cloud-based functions, all from different vendors). [22] describes a technology- and vendor-independent framework for the control and monitoring of network security functions and traffic. In the context of NFV, these network security function would be security-related VNFs. An I2NSF Controller would serve as an abstraction layer between the network security functions and the users (i.e., management systems).

5.5 Tungsten Fabric

Tungsten Fabric is the new version (5) of Juniper’s Open Contrail platform, which is now part of the Linux Foundation at [23]. To further clarify, Contrail Cloud is a commercial supported product from Juniper, whereas Open Contrail was a community supported project. When trying to install Tungsten Fabric in a self-hosted (as opposed to a public cloud-hosted) test environment, we ran into multiple failures. The documentation is not yet fully migrated to Tungsten Fabric, and refers Open Contrail multiple times.

Furthermore, we wanted to evaluate the integration of the Kubernetes (k8s) platform, as Juniper described in [24] that version 5 uses a micro-service architecture based on containers. However, after many tries we finally realized Tungsten Fabric still relies on OpenStack (e.g., the installation showed an error which indicated it couldn’t find OpenStack’s Keystone identity service). The documentation in [25] and [26] was not clear on this, as it specified only k8s could be used. Considering the complexity and different focus of OpenStack, namely data centers (as opposed to ISP networks), as stated in [27], it was decided to change the approach to a fully cloud-native infrastructure, and research other technologies than Tungsten Fabric.

5.6 Cloud-native infrastructure

Cloud-native technologies are described by the Cloud Native Computing Foundation (CNCF) in [28] as follows: "Cloud native technologies empower organizations to build and run scalable applications in modern, dynamic environments such as public, private, and hybrid clouds. Containers, service meshes, microservices, immutable infrastructure, and declarative APIs exemplify this approach. These techniques enable loosely coupled systems that are resilient, manageable, and observable. Combined with robust automation, they allow engineers to make high-impact changes frequently and predictably with minimal toil. The Cloud Native Computing Foundation seeks to drive adoption of this paradigm by fostering and sustaining an ecosystem.
of open source, vendor-neutral projects. We democratize state-of-the-art patterns to make these innovations accessible for everyone.”

When comparing this definition to the NFV objectives of ETSI, as stated in Section 2.1, we observed a certain overlap. Namely, we observed the open source nature, flexibility, and software-based service deployment characteristics of cloud-native infrastructure. As such, cloud-native infrastructure seemed appropriate to facilitate NFV. However, because of the result mentioned in the previous section, we proceeded with other technologies than Tungsten Fabric. Following this, we wanted to leverage cloud-native infrastructure to find the implications of virtualizing the security-related network functions mentioned in Section 5.1 and Section 5.2.

Figure 3 shows a cloud-native technology stack that we used to create a PoC, demonstrating the feasibility of using cloud-native infrastructure to provide VNFs. The sections below further explain the technologies in this stack.

5.6.1 Kubernetes

As stated in [29], k8s originated at Google as the Borg system, which is a cluster management system that executes Google’s applications. k8s was designed by many of the same engineers who created Borg, and accounts for the lessons learned with Borg. Furthermore, k8s was made open source and donated by Google to the CNCF. With Kubernetes we can deploy containerized (i.e., operating-system-level virtualization) applications across multiple hosts, and manage aspects like automated scaling and automated readiness checks. As further described in [30], k8s orchestrates the computing, networking, and storage infrastructure(s), in order to provision the applications. Moreover, k8s is a highly-modular platform, using plugins to interact with the aforementioned compute, networking, and storage infrastructure(s). Figure 4 gives an overview of k8s’ architecture.

According to [32], Pods are the smallest deployable objects in k8s; thus k8s manages pods and not containers, although the pods are made up of containers. Additionally, as described in [33], a k8s cluster is able to run on top of, but not limited to, an x86_64, ARM, or PowerPC operating system (e.g., Ubuntu, CentOS, Microsoft Windows, etc.). Furthermore, as mentioned in [34], a k8s cluster consists of two type of hosts: a master and a slave (also referred to as a minion or worker). The master host controls the cluster, and executes the master components/applications needed to provide the cluster’s control plane, whereas the slave host solely executes the user-specified applications. Moreover, it is possible to run multiple master hosts (i.e., a multi-master cluster) to achieve high availability of the k8s cluster control plane.

[34] further describes the individual master components/applications, which we will briefly summarize. An apiserver exposes the Application Programming Interface (API) and serves as the “front-end” for the k8s control plane. etcd is used as a highly-available key-value store, storing all cluster (configuration) data. A scheduler selects the host on which an application is executed, and a controller manager manages the controllers in charge of moving the cluster’s current state to the desired state (e.g., a replication controller manages the desired number of application instances).

Besides the master host, [34] additionally outlines the node components, that is, the applications that are used to manage every k8s cluster host. These applications are active on both the master and slave hosts. The applications consist of kubelet, kube-proxy, and a container runtime. The kubelet is an agent in charge of running the containers, and the kube-proxy is in charge of the networking rules and traffic forwarding. The container runtime is the software that actually runs the containers, and is called by kubelet. Section 5.6.3 further describes the container runtime. Finally, the next section further details the networking aspects of k8s.

5.6.2 Multus

As stated in [35], the Container Network Interface (CNI) is part of the CNCF and concerns itself with the configuration of network interfaces in Linux containers. The CNI is supported by k8s. Furthermore, as is described in [36], Multus is a CNI plugin for k8s created by Intel. Using Multus, it is possible to allocate multiple interfaces to individual pods (i.e., multithoming). Additionally, Multus functions as a self-described meta-plugin, referring to the capability of calling multiple other CNI plugins. Thus, by leveraging Multus, we are able to use multiple CNI plugins per pod. This allows for the provisioning of multiple networks, each with their own characteristics respecting security and performance segregation. Moreover, other CNI plugins to choose from could be overlay (i.e., tunnel inter-pod communication
traffic) or underlay (i.e., route inter-pod communication traffic) network plugins. Figure 5 illustrates the Multus CNI being used to create three separate networks in a pod.

![Network Functions Virtualization and Security](network-functions-virtualization-and-security.png)

Figure 5. The Multus CNI being used to attach a k8s pod to three different networks. Source: [36]

When further looking at the details of k8s networking, we observed in [37] and [38] that k8s assigns an IPv4 address to each pod. Additionally, the containers within a pod share all the interfaces of the pod. Within the Linux kernel, this is achieved by creating a network namespace (netns) per pod, and placing all containers within the same netns. However, in order to achieve inter-pod communication, services are used within k8s. A service can be seen as an entry point for a pod or collection of pods. A service will have an IPv4 address assigned and will load balance the traffic to its assigned pods. Figure 6 illustrates this concept. Furthermore, in order to discover a service, environment variables or the Domain Name System (DNS) can be used (e.g., an application, running in a pod, needs to access another application, also running in a pod, via a service, using the service’s DNS name).

By default, services are using a ClusterIP, which makes the service accessible on a cluster-internal IPv4 address. However, as further stated in [38], should outside access to a service be required, the service can be published using a NodePort or LoadBalancer. The first entails the usage of the k8s host IPv4 address, in order to bind a static port that will route to the service. The latter entails the usage of a cloud providers load balancer. When looking back at Figure 5, we can observe that the pod’s eth0 interface is assigned the cluster-internal IPv4 address.

Last, as further described in [38], kube-proxy uses the Linux Kernel’s netfilter framework in order to redirect the traffic destined for the service IP address, to one of the pod IP addresses instead.

5.6.3 CRI-O

[39] states that CRI-O is a lightweight container runtime for k8s. Specifically, CRI-O implements the Container Runtime Interface (CRI), which allows k8s to support multiple container runtimes. CRI-O was specifically designed for k8s and removes the need for using Docker as the container runtime, thereby reducing the attack surface, since Docker has a larger scope beyond supporting k8s. CRI-O is open source and developed by multiple companies, including Red Hat and Intel. Kubernetes’ kubelet uses the CRI to communicate with a container runtime to execute a container, as is described in [40].

Furthermore, CRI-O supports runtimes that adhere to the Open Container Initiative (OCI). As stated in [41], the OCI specifies open standards around container image formats and runtimes. The OCI is part of The Linux Foundation. CRI-O is thus a bridge between k8s CRI, and the OCI-compliant runtimes, thus CRI-O is more of a container runtime manager, rather than an actual container runtime itself. The default OCI runtime used to create containers is runC. Once a container is created, a process named common is used to monitor the container. Moreover, CRI-O makes use of the CNI to establish the container networking aspects. Figure 7 gives an overview of CRI-O’s architecture.

![CRI-O architecture](cri-o-architecture.png)

Figure 7. CRI-O architecture. Source: [39]

5.6.4 Kata Containers

As stated in [42], Kata Containers is an open source project aimed at using lightweight VMs that handle as containers (i.e., each pod is a VM), but with the added security advantages of VMs. Furthermore, Kata Containers is compatible with multi-
ple hypervisors, in addition to the OCI and CRI. The project is the result of a merger of Intel’s Clear Containers project and the Hyper runV project, and is part of The OpenStack Foundation. Figure 8 gives a high-level overview of Kata Containers.

Figure 8. Kata Containers architecture. Source: [42]

When further looking at the architecture of Kata Containers, we can observe its integration with CRI-O and k8s. Figure A2 in Appendix B gives a detailed overview of k8s runtimes, including its inner workings with Kata Containers. Specifically, we observed there are multiple ways to launch workloads in containers or VMs. From kubelet, a container runtime manager, e.g., CRI-O, Docker (using the docker-shim), or containerd (which is also leveraged by Docker) will first be executed using the CRI, then the actual container runtime, i.e., runC for containers or kata-runtime for VMs, will be called using the OCI specification.

We can differentiate between the runtimes using k8s’ RuntimeClass alpha feature, as described in [43]. Furthermore, there are multiple in-between processes being used to facilitate the management of a container once it has been created (e.g., containerd-shim, common, and kata-shim). Moreover, Kata Containers recently was able to reduce these in-between processes to just one (i.e., containerd-shim-kata-v2) per pod, as described in [44], reducing the complexity. Unfortunately, at the time of writing this reduction was only compatible with containerd, and not CRI-O. Finally, regarding the VMs, Kata Containers makes use of a kata agent, which is running inside of each VM, and is responsible for creating the actual containers.

5.6.5 NEMU
No EMUlation (NEMU) is a hypervisor build by Intel for cloud-specific workloads. As described in [45], the hypervisor is a fork of the Quick EMUlator (QEMU), but with all of its non-cloud-related features removed, thereby lowering the attack surface and achieving higher performance. Specifically, Intel introduced a new x86_64/ARM machine type, virt, which adheres to the hardware-reduced ACPI specification. This specification is designed for UEFI-platforms only, thus removing legacy support. NEMU, like QEMU, makes use of the Kernel-based Virtual Machine (KVM), which results in a hardware-assisted full virtualization platform.

5.7 Proof of concept
Figure 9 gives a high-level overview of the PoC, whereas Figure A3 in Appendix C gives a detailed overview. The following software versions were used:

- Kubernetes v1.12.2
- Multus master branch as per Feb. 18, 2019 (v3.1 is missing patches)
- CRI-O v1.12.2
- Kata containers v1.4.0
- runC v1.0.0-rc5
- NEMU/KVM: latest, v3.1.0 as per Feb. 12, 2019
- k8s host VM: CentOS 7.6 (kernel v3.10.0-957.5.1.el7.x86_64)
- QEMU/KVM hypervisor v1.5.3 (CentOS 7.5)

These versions were the result of choosing the Kata Containers version. [46] describes the dependencies for Kata Containers version 1.4. Version 1.5 was the most recent at the time of the research, but the integration with k8s did not support CRI-O using shimv2. The underlying hardware consisted of a Dell PowerEdge R230 x86_64 server (using an Intel Xeon E3-1240L CPU, and 16GB of memory).

Figure 9. High-level overview of the proof of concept.

The PoC consisted of two k8s nodes: a master and slave. Both nodes were capable of running all workloads (i.e., k8s control plane pods and user-specified applications, using runC and NEMU). The nodes were connected to two different networks: a control plane and a data plane network. The control plane was used for regular network connectivity such as remote management of the nodes themselves, outgoing traffic to the Internet, and to facilitate inter-pod communication by using the Flannel CNI, which creates an overlay network using the Virtual Extensible LAN (VXLAN) protocol. The data plane network was used to simulate the fabric where the actual live traffic is being exchanged between the educational institutions (i.e., SURFnet’s network).

Additionally, Figure 10 gives an overview of a self-built NFV Controller application.

The NFV Controller was built using the Flask Python micro web framework, [47], and the rqlite distributed relational database, [48]. The application was able to automatically register VNFs as they came online, and remove them when they terminated. Furthermore, it was possible to insert information (e.g., IPv4 addresses) about educational institutions, and to create rules that would “direct” traffic to a VNF. The application was also able to scale automatically, and independently, as per the micro services architecture.

Regarding the exact implementation of the traffic manipulation and VNF, we created a simple shell script, and based our endpoints on IPv4 addresses. The NFV Controller would call the shell script to insert a rule, using the iptables command, into the hypervisor that’s running the data plane. This rule would then duplicate all traffic to the IPv4 address of the VNF. The VNF was implemented using the tcpdump...
packet analyzer, and would log all traffic to a file inside of the VNF pod. The NFV Controller pod was executed on top of runC, whereas the VNF pod was executed on top of NEMU/KVM. Thus we succeeded in creating a simple example of a VNF leveraging cloud-native infrastructure. The location of the source code and the k8s configuration files of the NFV Controller and VNF is listed in the appendices.

5.7.1 PoC example run
In this section we will look at an example run of the PoC. Listing 1 shows the output of starting the deployment of the NFV Controller and VNF.

Listing 1. Deployment of the NFV Controller and VNF.
```
[root@master rp2] # ./k8s_deploy_all.sh apply
Starting k8s deployment script!
Deploying...
 rqlite database cluster...
 service/rqlite-cluster created
deployment.apps/rqlite-leader created
deployment.apps/rqlite-node created

SURF NFV Controller...
deployment.apps/surf-nfv-controller created
service/surf-nfv-controller created
service/surf-nfv-controller-nodeip created
Init DB
Database initialized.
VNF tcpdump...
deployment.apps/surf-vnf-tcpdump created
Inserting test values...
Inserting edu
123
```

Listing 3 shows an NFV Controller instance receiving the VNF registration using the control plane network.

Listing 3. VNF registration at the NFV Controller.
```
[["class": "FaaS", "id": 1, "ip":
  "10.200.0.103"]]
```

Please insert rule and generate traffic.
Done!
```
```
```
```
```
```

Figure 10. Overview of the NFV Controller application.

Figure 11 shows the running pods of the PoC. We can clearly observe the multiple instances of the same pods.

Figure 11. Overview of the PoC deployment (visualization tool: [49]).

In Listing 2 we further see the insertion of a rule. This rule specified that traffic from IPv4 address ‘10.200.0.11’ (the ‘UvA’ educational institution, as earlier listed above) had to be duplicated to VNF ip address ‘10.200.0.103’ (the ‘FaaS’ VNF, as earlier listed above). Note that these addresses belonged to the data plane network, and not to the control plane network. The VNF used a different path, namely the k8s-managed one, to communicate with the NFV Controller, which did not have a presence in the data plane network itself (but could control the hypervisor, which did have a presence). IPv4 address ‘192.168.122.17’ was the address of the k8s master node, and the port is managed by k8s to be forwarded to one of the NFV Controller instances.

Listing 2. Insertion of a rule in the NFV Controller.
```
[root@master rp2] # curl --header
   "Content-Type: application/json"
  \n
  > --request PUT

  > --data '{"edu_ip":"10.200.0.11"
   "vnf_ip":"10.200.0.103"}'


```

Listing 3. VNF registration at the NFV Controller.
```
* SURF NFV Controller
* Database server: rqlite-cluster (rqlite)
* Serving Flask app "nfv_controller" (lazy loading)
* Environment: production
Furthermore, Listing 4 shows the traffic duplication rule the NFV Controller inserted in the hypervisor, where the IPv4 addresses of the educational institutions live.

Listing 4. The rule the NFV Controller inserted on the hypervisor / data plane network.

```
Chain PREROUTING (policy ACCEPT)
target TEE
proto all
opt --
source 10.200.0.11
destination 0.0.0.0/0
TEE gw: 10.200.0.103
```

Finally, when we generated some traffic from the educational institution to a destination within the data plane network, we observer the successful arrival of the traffic at the VNF, as is shown in Listing 5. Note that we are only duplicating traffic originating at the educational institution, but not the returning traffic.

Listing 5. Arrival of traffic at VNF.

```
No. 1
Time 0.000000
Source 10.200.0.11
Destination 10.200.0.1
Protocol ICMP
Length 98
Info Echo (ping) request id=0x3b48, seq=1/256, ttl=63 (no response found!)
```

However, when we changed the VNF’s runtime from NEMU to runC, we observed traffic being sent from the VNF back to the educational institution (‘10.200.0.104’ is the VNF’s IPv4 address), as is shown in Listing 6.

Listing 6. VNF doing an ICMP redirect.

```
No. 65
Time 0.999992
Source 10.200.0.104
Destination 10.200.0.11
Protocol ICMP
Length 126
Info Redirect (Redirect for host)
```

6 DISCUSSION

In this section we will analyze and evaluate our results, and list the limitations of our research.

6.1 Tungsten Fabric

As observed in our results, Tungsten Fabric still relies on OpenStack. Considering Tungsten Fabric is also described, in [37], as a CNI plugin, this raises the question regarding the complexity of the overall infrastructure if both k8s and OpenStack are required.

6.2 Kubernetes

The utilization of the k8s clustering system might result in extra infrastructure to maintain. However, when k8s is used for other workloads besides NFV, we achieve a reduction in infrastructure complexity, because the same platform can now be leveraged in order to provide a wide range of virtualized functions (e.g., identity management, intranet portals, asset management systems, network functions, etc.). The modular nature of k8s further allows for the tailoring to specific use cases, as we observed with the usage of Multus. From a security perspective, this (i.e., leveraging the same platform for diverse workloads) has the further advantage of reducing the infrastructure components that have to be audited, and hardened.

However, when realizing our PoC, we observed one has to be skilled in the areas of Infrastructure as Code (IaC), the micro services architecture, and the overall DevOps methodology. These aspects do indeed manifest a paradigm shift in network engineering. As such, any organization trying to leverage the benefits of this methodology, would first have to familiarize its engineers with this change in approach.

Additionally, we can observe k8s has multiple building block to realize VNFs, but these block require extensive manual tooling and in-depth knowledge of the k8s platform. Finally, we observed the lack of building blocks regarding the networking aspects of the pods, as further described in the sections below.

6.2.1 Network plugins

Whereas our results show that k8s can be used to manage the VNFs’ network as far as the control plane is considered, the data plane, however, remains a challenge. Within our PoC we managed our NFV Controller using k8s, but the data plane itself was managed outside, or rather on top of, k8s (e.g., for the traffic steering we used functionality outside of the k8s cluster, namely our shell script via the NFV Controller, instead of a k8s service). To further clarify, within our setup, k8s was serving all three functions of the NFV-MANO working domain: the VIM, the VNFM, and the NFVO. However, due to the very nature of NFV, we need to have full insight into the inner workings of the networking aspects regarding the pods. This seems to be add odds with the design of k8s, which facilitates the abstraction (i.e., hide) the networking aspects away from the user, and is focusing on providing a black box that is capable of running multiple networked applications.

What we need is a white box approach regarding the networking aspects, since our applications are in fact implementing network functionality. Specifically, we are implementing similar network functions in pods, that are currently
assigned (and abstracted away) to the kube-proxy component. In summation, regarding using the pods to implement the VNFs, we are changing the pods location in the network from an endpoint to a transfer point. As such, we break k8s design, as its CNI plugins assume the pods are endpoints.

6.2.2 Container runtime
When we look at the number of components k8s needs to execute a workload, we wonder if we cannot integrate these components further. E.g., if k8s would natively support the OCI specification, there would be no further need for in-between components, and thus simplify the overall architecture. Following this observation, a major point of interest is leveraging the capabilities of the systemd platform. Specifically, systemd supports the creation of containers and VMs, as stated in [50] and [51], thus it might be beneficial of using systemd as the container/VM runtime, as it is already present in most Linux distributions. The rkt container engine, developed by CoreOS, seems to make use of this architecture, as described in [52]. However, it is unclear if rkt is still being actively maintained, since Red Hat’s acquisition of CoreOS.

6.3 VNF placement scenarios
When we look at the utilization of a k8s cluster to provide NFV, we can distinct several scenarios. Figures 12 through 15 illustrate these scenarios. Two major considerations are whether the k8s cluster will be installed in-line, i.e., within data path, and whether the cluster will only be utilized to tap traffic, or will also forward traffic. As earlier mentioned, k8s does not ready facilitate the implementation of network functions in pods (i.e., transfer/route traffic within pods). As such, it is recommended to focus any further efforts on the experimentation with tapping/duplication of traffic to an out-of-line k8s cluster, rather than replacing the current forwarding fabric with a k8s cluster.

6.4 PoC behaviour regarding the VNF runtime
As to why the VNF was sending traffic back to the educational institution, we are unable to answer in depth. However, we assume this is related to the fact that, when using runC, both the source and destination IP address are active within the same kernel, albeit using different network namespaces. Because of this observation, we can attest that the usage of operating-system-level virtualization might result in insufficient isolation, hence the need for VMs. Furthermore, this result shows the inherent complexity of developing VNFs, and the need for a white box approach, because the VNFs might behave differently based on the underlying technology.

6.5 Limitations
Because of the modular nature of k8s, and the many technologies (i.e, plugins) that support k8s, we limited our research to relatively new k8s-related projects that were done by Intel Corporation, namely Multus, Kata Containers, and NEMU.

Furthermore, regarding the NFV Controller of the PoC, load balancing between multiple VNFs is not possible. We also assumed a unique relation between each educational institution and VNF, that is, a VNF can not be assigned to multiple educational institutions. However, it was not the focus of this research to evaluate the technologies available for traffic steering (i.e., creating a VNF-FG). Instead, we looked at how the VNFs themselves might be implemented in relation to the virtualization technology used, and the implementation of the control plane.

Finally, we obviously limited ourselves to cloud-native technologies, and specifically the k8s cluster platform. As such, we might observe far different results when utilizing, for example, proprietary and/or non-cloud-native NFV platforms.

7 Related work
Multiple studies have been performed on the aspects of NFV security. [53] gives a comprehensive overview of the state-of-the-art in NFV, and specifies several research challenges regarding NFV. The study specifically mentions security, privacy, and trust being major topics of possible research. [54] gives a brief overview of the challenges and opportunities in NFV
security, and looks at products from Cisco and VMware. In addition, [55] looked at existing and possible attacks against both NFV and SDN, and listed mitigation techniques. [56] looked at security threats and provided countermeasures as well. Furthermore, [57] included a threat analysis in the context of NFV, and provides a framework for NFV-based security management and service orchestration.

[58], [59], and [60] also looked at utilizing cloud-native infrastructure for NFV, but focused on 5G and using the VNFs as endpoints rather than transits. In particular [58] focused on using containers only with Kubernetes, and did not look at VMs. Furthermore, [59] mentions k8s lacks clear support for traffic steering between VNFs (i.e., via a VNF-FG). However, the work mentioned Multus being an option to mitigate this. The work also considered containers to lack maturity regarding the isolation of VNFs. Moreover, [60] describes a not yet completed study.

The added value of the research described is this paper was to extend on the previous work done at SURFnet by including the security aspects of NFV, and to examine the feasibility of leveraging cloud-native infrastructure in order to provide the VNFs.

8 CONCLUSION

In this paper we looked at the feasibility of ISP-provided security-related network functions to the campus network of educational institutions, by making use of NFV. We first looked at an overview of security-related network functions in campus networks, and then observed which functions could be provided by the usage of NFV. Furthermore, we looked at the state-of-the-art of NFV security work done by ETSI, and listed several security implications inherent to the NFV technologies/platforms themselves. Additionally, we set out to evaluate the Tungsten Fabric platform, but we concluded that the platform’s documentation and focus were not suitable for our use case. We therefore decided to look at leveraging cloud-native infrastructure, as it aligns with the goals of NFV. Consequently, a PoC was built by making use of cloud-native technologies like Kubernetes, and further augmented with a self-built NFV Controller. Finally, we discussed the implications of using NFV to realize the security-related network functions, observing several challenges regarding the placement of the functions within the ISP’s network.

In conclusion, cloud-native infrastructure has a lot of potential, but lacks the maturity to offer security-related network functions for campus networks. We need further integration of existing and upcoming technologies in the areas of configuration, orchestration, and overall management, as to provide a turn-key solution. Note, however, that this conclusion is based on the assumption that we require a vendor-independent, and open source solution, as per the NFV objectives defined by ETSI. In relation to vendor-specific, and possible closed source, solutions, SURFnet might be able to realize its offering of security-related network functions to the campus networks, albeit without the advantages of vendor independence.

9 FUTURE WORK

Future work might include the research of several virtualization technologies that can enhance and/or further enable NFV. Specifically, the Open Source Networking projects listed under the Linux Foundation at [61] are of great interest. Two main projects of interest are OPNFV and ONAP. Both of these projects focus on the integration of NFV components. The following three technologies are also of interest, when related to the research presented in this paper: the extended Berkeley Packet Filter (eBPF), DPDK, and Vector Packet Processing (VPP). All these technologies deal with the offloading of network processing from the Linux kernel (i.e., the netfilter framework), in order to achieve higher networking performance. As such, these technologies make for an ideal candidate to enable the data plane functionality. Security and performance testing would thus be areas of interest.

Additionally, ClickOS, as described in [62], might be of interest. ClickOS implements NFV by using heavily-optimized VMs running on top of the Xen hypervisor. Comparisons with the earlier mentioned technologies above (e.g., QEMU/KVM and eBPF), in addition to the integration with cloud-native technologies might give us further insights into the possibilities of NFV.

Furthermore, even though Tungsten Fabric did not meet SURFnet’s requirements in its current form, future releases might provide different functionality. It is therefore recommended to revisit this platform in a year or so.

Finally, when we look beyond the technical aspects of NFV security, the legal and ethical aspects of ISP-provided security-related network functions might also be of major interest. For example, considering SURFnet has to adhere to the EU’s General Data Protection Regulation (GDPR), there might be major considerations that further influence the functions’ design (e.g., what is being logged?, by whom?, where?, and how?).

REFERENCES


ACKNOWLEDGEMENTS

The author would like to thank SURFnet for providing and supporting this research subject.

APPENDICES

The source code of the Proof of Concept (PoC) can be found at the following GitHub repository: https://github.com/tiredcoder/surf-nfvsec. The source code includes the Kubernetes deployment files, the NFV Controller application, and the build files needed to create the container images. The container images can also be found at the Docker hub (latest builds): https://hub.docker.com/u/rjos3. Furthermore, the GitHub repository also includes the steps that are necessary to create the PoC (e.g., software installation and configuration), and the test results (the network traces).

The following appendices are included with this paper:

A. VNF-FG
B. Kubernetes runtimes
C. Detailed overview of the proof of concept
APPENDIX A
VIRTUAL NETWORK FUNCTION FORWARDING GRAPH

Figure A1. Physical View of a VNF-FG. Source: [8]
Appendix B
Kubernetes Runtimes

Figure A2. Kubernetes runtimes. Adapted from: [44]
APPENDIX C
DETAILED OVERVIEW OF THE PROOF OF CONCEPT

Figure A3. Detailed overview of the proof of concept