Container-based design of a Virtual Network Security Function

Marco De Benedictis
Politecnico di Torino
Dip. Automatica e Informatica
Torino, Italy
Email: marco.debenedictis@polito.it

Antonio Lioy
Politecnico di Torino
Dip. Automatica e Informatica
Torino, Italy
Email: lioy@polito.it

Paolo Smiraglia
Politecnico di Torino
Dip. Automatica e Informatica
Torino, Italy
Email: paolo.smiraglia@polito.it

***Abstract*—Modern ICT infrastructures are evolving thanks to the advantages offered by virtualisation in terms of flexibility, scalability, and savings on hardware-related costs. More recently, virtualisation has gained momentum in the Internet Service Providers' infrastructures as well, where Software Defined Networking and Network Function Virtualisation paradigms propose programmability of the network and the softwarisation of proprietary hardware appliances. In this scenario, lightweight virtualisation technologies, such as Linux containers, have a significant role, as they address the needs for scalability, availability and fast deployment to support the software-based network infrastructures. In this paper, we focus on defining a reusable design for a container-based Virtual Network Security Function, by highlighting the peculiarities of its architecture compared to a Virtual Machine-based instance. Moreover, we present a prototype application of this architecture to implement an HTTP reverse proxy with application-layer filtering capabilities, tailored for the NFV Security-as-a-Service scenario. We evaluate the performance of this prototype and compare it to the results of alternative deployments, namely the Virtual Machine and bare-metal solutions. Finally, we evaluate the proposed solution in a load-balancing scenario, for increased throughput and availability.**

I. Introduction

Internet Service Providers (ISPs) are experiencing a paradigm shift in their networks, due to flexible networking technologies such as *Network Function Virtualisation* (NFV) and *Software Defined Networks* (SDN). More specifically, the network operators benefit from the advantages of virtualisation in terms of availability, scalability and flexible resource usage. *Virtual Network Functions* (VNFs) are deployed in the operator infrastructure to deliver both networking and security services for the provider itself and the end users as well. The conjunct use of NFV and SDN enables ISPs to instantiate network security services close to threats, in order to minimise the reaction time and maximise the mitigation result.

In this scenario, the *Security-as-a-Service* (SecaaS) paradigm can be exploited by ISPs to adopt security-oriented VNFs, i.e. *Virtual Network Security Functions* (vNSFs), that can be provided to their clients, such as enterprise networks.

Virtual Machines (VMs) have been traditionally the target for the NFV reference implementations, although different NFV frameworks are moving towards lightweight virtualisation techniques recently. In particular, *Linux Containers* implement *Operative System* (OS) virtualisation, allowing a single host to run different processes in isolated sandboxes that share the same kernel. This technology allows faster deployment than traditional VMs, as containers virtualise the user-space portion of the OS only, hence they are lighter than traditional virtualised instances. Moreover, the overall density of applications running on the same host can be higher on container-based virtualisation, as part of the OS is shared with the host. From the security perspective, containers leverage Linux kernel technologies, namely *Cgroups*, *Root Capabilities* and *Namespaces*, to isolate the different virtual instances and allocate resources for each of them.

Although they provide significant advantages over traditional virtualisation, containers have not been yet fully exploited for the NFV scenario. This paper aims at tackling this gap, proposing a concrete, replicable design of a container-based vNSF, which leverages the interaction between lightweight virtualisation environments for its configuration, reporting and security application. In this regard, the authors also present a reference implementation for the container-based vNSF, highlighting its peculiarities in respect to a VM-based deployment. This prototype is evaluated in a specific application of the SecaaS use case, which is targeted for the protection of a web application. The authors do not intend to propose a security analysis on container-based vNSFs in this work, although security-oriented principles and best practices are considered as part of the overall design and implementation of the prototype.

The paper is structured as follows. The related work on this subject is described in Section II. Then, the motivation behind this research is presented in Section III. Section IV describes the SecaaS scenario where the proposal is developed, followed by the design and architecture of the container-based vNSF in Section V. The prototype of the vNSF, that implements a reverse proxy enhanced with application layer filtering, is presented in Section VI. Then, experimental results of the evaluation of the prototype are reported in Section VII. Future work on the proposed solution is presented in Section VIII. Finally, the authors' conclusions are reported in Section IX.

978-1-5386-4633-5/18/$31.00

II. Related Work

Containers have been recently addressed by open-source reference implementations of the NFV architecture. The *Open Baton* [1] framework implements an ETSI compliant NFV *Management and Orchestration* (MANO) stack, and its latest release at the time of writing provides built-in support for a specific container engine, named Docker [2], in addition to an *OpenStack* cloud management system. Docker is a widespread lightweight virtualisation platform, which defines both the packaging specification of a container, via the so-called Dockerfile, and its running environment, the *Docker Engine*. It also defines the requirements and work-flows to ship and distribute containers via centralised repositories such as the built-in *Docker Hub*. Although different container solutions exist, Docker has quickly become the de-facto standard for production environments, being supported by Amazon Web Services, Microsoft Azure and Google Cloud Compute Engine. Open Baton has recently implemented a MANO *Virtual Infrastructure Manager* (VIM) driver for Docker. The *OPNFV* [3] framework leverages Open Baton for the MANO stack and it supports both OpenStack for standard VMs and *Kubernetes*, a production-ready container management system that can run different lightweight virtualisation run-times, including Docker. The platform developed by the *Open Source MANO* (OSM) [4] project, an ETSI-hosted activity based on the ETSI NFV specifications, does not fully support containers at the VIM level at the time of writing, although the *VIM emulator* tool [5] has been proposed recently to emulate a VIM using Docker containers.

Application of lightweight virtualisation techniques to NFV has been investigated in literature as well. Cziva *et al.* [6] present a framework, named *Glasgow Network Functions* (GLANF), to manage the life-cycle of VNFs in a *OpenFlow-based* SDN infrastructure, leveraging Docker to achieve low performance overhead and to reduce deployment time. The authors' evaluation of their Docker-based network functions shows that the instantiation time improves by up to 68% over hypervisor-based virtualisation techniques, such as Linux KVM or Xen. Cziva and Pezaros [7] also propose a concrete application of GLANF to the network edge, exploiting lightweight container-based network functions that can run on a variety of edge devices. Anderson *et al.* [8] investigate performance of container-based network functions, addressing Docker as a promising technology to enhance efficiency and deployment time of NFV environments. The authors also stress the potential of Docker in terms of packaging specification, as a means to standardise the vNSF format and to ease its distribution. The authors discuss the performance overhead introduced by Docker standard networking technologies, namely *Linux bridge* and *Open vSwitch*, and evaluate alternative solutions, such as *macvlan* and *SR-IOV*.

The Docker container engine is gaining momentum both in the literature and the technical solutions regarding the NFV environment, because of its potential to address relevant challenges in terms of performance and distribution of vNSFs. Hence, the research described in this paper focuses on Docker for the implementation of a container-based vNSF prototype.

III. Motivation

The ETSI NFV Industry Specification Group has developed a standard [9] to discuss applicability of different virtualisation layers to the NFV infrastructure. The hypervisor-based technologies are considered the present typical solutions for the deployment of VNFs, hence ETSI has developed an hypervisor-domain specification [10] to clarify the interaction between the hypervisor, the VIM and the compute nodes that host the virtual machines. OS virtualisation is considered as an alternative for the deployment of VNF Components [9] in case multiple instances of the same VNF need to be deployed on the same ISP infrastructure, but more specific details on the container-domain in NFV are missing at the time of writing. Moreover, the high-level architecture of a VNF [11] encompasses both VMs and containers to implement the VNF building blocks, but does not differentiate among peculiar design principles of the different virtualisation techniques. The authors aim at defining a reusable design for a container-based vNSF, which is currently missing from literature. Available frameworks for the NFV focus on orchestration of network functions in both hypervisor and container domains, although they do not provide concrete specifications on the internal structure of the VNFs to be deployed on top of these domains.

IV. Scenario

The design and development of a container-based vNSF is framed in a specific use case of the ETSI NFV framework, i.e. the *Security-as-a-Service* [12] scenario. This use case is motivated by the need of protecting ICT infrastructures effectively in an evolving threat environment, where vNSFs can be exploited to both monitor and react upon detected attacks. An ISP could build SecaaS services to secure his clients' networks, freeing them from the costs of managing, operating and upgrading dedicated network and security devices. In this scenario, big data analytics can play a significant role to fulfil the anomaly detection and define a mitigation strategy.

A high-level architecture of the SecaaS use case is depicted in Figure 1. It includes the following components:

- ***Data Analysis & Remediation Engine* (DARE).** It performs threat detection using analytics, cognitive intelligence and monitoring of the infrastructure, in order to define a mitigation strategy for an attack.
- **vNSF store.** A catalogue of vNSFs that can be instantiated in the network, belonging to different categories:
  - **Monitoring vNSF.** It monitors the traffic of the network, acting as a network probe, event generator or honeypot. The relevant information of monitored traffic is fed to the DARE.
  - **Reaction vNSF.** It applies a mitigation strategy defined by the DARE, with the aim of preventing or stopping a threat.
- **Dashboard.** A visualisation component that can be accessed by the infrastructure administrator, to display the

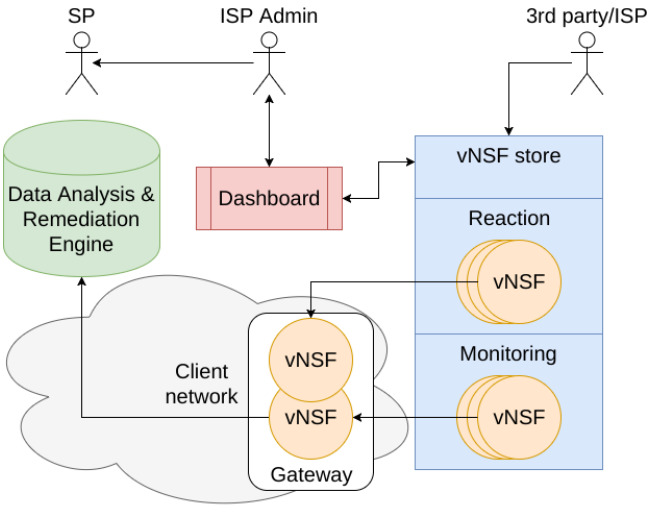

Fig. 1. ETSI NFV Security-as-a-Service use case

analytics result and recommend mitigations for incoming threats.

The vNSFs can be deployed onto the client gateway or in the ISP network infrastructure. In addition to these components, the SecaaS use case also specifies vNSF and infrastructure attestation as a necessary step to ensure that the SecaaS service is trustworthy. In this scenario, the design and development of a container-based vNSF should take into account the need for providing monitoring and/or reaction capabilities for specific security threats, in addition to reporting the relevant network and security information to trusted third-parties for analytics.

The SecaaS use case introduces relevant challenges in terms of correctness, availability and scalability of the vNSFs, given their critical role in a ISP infrastructure. These requirements are typically implemented in traditional deployments of security middle-boxes, hence they should be considered by the NFV paradigm as well. The authors propose a design that addresses both availability and scalability requirements by focusing on a lightweight virtualisation technology, which provides both reduced deployment time, more flexible reconfiguration and consolidation. The correctness requirement, which would ensure that the vNSF application acts as expected, should be addressed by means of attestation and/or monitoring of the vNSFs ecosystem, hence it is not targeted explicitly by this proposal.

V. Design and Architecture

The high-level design of a vNSF is depicted in Figure 2. A single vNSF is composed by different *vNSF Components*, which can be implemented by VMs or containers, according to the standard [11]. In our proposal, the vNSF Components are to be implemented by different containers, which are by nature stateless virtual instances. In order to share data among different containers, OS-level virtualisation proposes both standard network interaction and *volumes*. Volumes define a *binding* between a local directory of the container and another path,

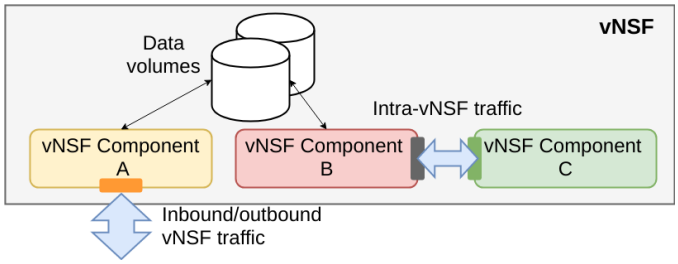

Fig. 2. vNSF design

external to its mount namespace. The Docker container engine suggests the use of volumes because they do not require modification at the network configuration of containers and underlying hosts. Moreover, they can be stored on remote hosts or cloud providers, hence they are independent from the directory structure of the host machine. Docker volumes can be shared by containers running heterogeneous OSes and they do not add to the overall size of the virtual instance. For the sake of modularity, we base our initial design on volumes as they fulfil the requirements in terms of reliable interaction between containers and persistence of the application data. Moreover, they can be paired with *Mandatory Access Control* technologies, such as the *Security-Enhanced Linux* (SELinux) system in the Linux kernel, to manage access to file system object via enforceable security policies. Nonetheless, we consider network communication in-between containers as a viable alternative to volumes, as it would remove the overhead related to the file system access. However, it is to be noted that introducing network-based interaction between Docker containers would require additional client-server services to be run on each of the instances.

In a micro-service oriented design, single user processes are run by different virtual instances. This paradigm is embraced by OS virtualisation implementation design as well, according to the *one process per container* principle. This methodology aims at decoupling the different applications, to address scalability of the service and reuse of the implementations. Because of this, we expect to have more *intra-vNSF* communications in a container-based scenario than a VM-based implementation. In this regard, the network interaction would require each container to run additional processes to manage the client-server connection, interfering with this paradigm.

From a security perspective, the design of a vNSF should limit the number of containers that are accessible from the network. In Figure 2, only *vNSF Component A* manages inbound and outbound traffic, while the other components contribute in the fulfilment of the internal logic of the application. The vNSF traffic can be directed towards the external networks or to/from other vNSFs of the same network service. Container management systems use *port mapping* on the host to expose an internal port of a virtual instance. The attack surface of the host increases by adding port mappings to expose services deployed in the containers, hence it is necessary to minimise the number of open ports on the host.

Having the requirements in mind, the authors propose a detailed architecture for a vNSF, tailored for the SecaaS use case. In this regard, such virtual instance should be able to receive security-oriented policies to define monitoring and reaction actions to security threats, collect and report application logs to an external entity, for higher-level monitoring and further analysis and, finally, apply the vNSF logic by leveraging internal communication between highly-specialised containers. We propose a vNSF architecture that is composed of different containers to implement specific capabilities, which share information using volumes. There is a one-to-one mapping between each of these capabilities and a separate VNF Component. The architectural components are described as follows in this section.

vNSF volumes

As introduced before, volumes are a viable solution to share and persist information among different containers, even if they are deployed on separate hosts. In our proposal, three different volumes are introduced to both persist and share separate information:

- **Policy volume.** It is used to receive security policies that define the behaviour of the vNSF, and it is typically accessed with write permission by a management entity such as a *security controller/orchestrator* [13].
- **Configuration volume.** It is used to map the directories containing configuration files that are directly used by the vNSF application software.
- **Log volume.** It contains the logs of the vNSF application and consumed by the reporting logic for further analysis.

Policy translator

The ETSI NFV specification states that VNF may receive policies from the MANO stack via a specific logical interface. The proposed architecture includes a container to serve as a security policy translator. It is in charge of processing the input policies, which are received in the Policy volume, and to write specific configuration rules that can be applied by the vNSF application logic in the Configuration volume. This container is not conceived as a long-lived entity, as its execution terminates as soon as it writes the application-dependent policies on the Configuration volume.

The abstraction of a policy language serves as an intermediate layer to enable interoperability between different vNSFs implementations that tackle the same security control. For example, a SecaaS service could allow the user to choose between two different implementations for a Layer 3 filtering function, allowing vNSF developers to implement translation logics tailored for the filtering application (e.g. *iptables*, *pfSense*, *Open vSwitch*).

In this regard, we propose the reuse and extension of the policy abstraction language defined in the scope of the *SECURity at the network EDge* (SECURED) [14] FP7 European research project. It defines application-independent configuration rules for different security capabilities, specified in a *Medium-level Security Policy Language* (MSPL), that represent the building blocks of network and security controls [15]. The SECURED project also specifies a centralised process to translate policies to application-level configuration rules, pursued by a *Security Policy Manager* of the infrastructure. In our proposal, we offload the translation to the single vNSF, to reduce the load on the centralised NFV MANO stack.

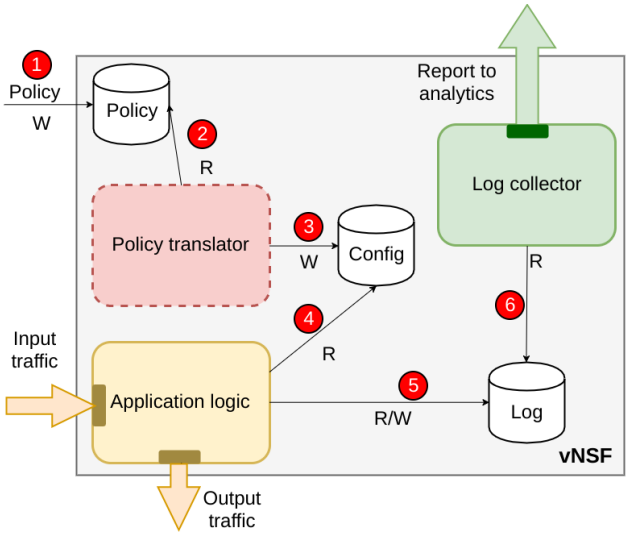

Fig. 3. vNSF architecture and instantiation work-flow

Log collector

This component is in charge of aggregating the output of the vNSF application logic and to package it in a format that is processable by third entities, such as the SecaaS Data Analysis and Remediation Engine. The logs are read from the Log volume, which serves as a persistence entity within the vNSF. It is to be noted that additional techniques are needed to ensure reliability and availability of application logs, especially for auditing purposes, and this component is not meant to fulfil the auditing requirements for a NFV environment. Logs are merely conceived as an evidence of the monitoring/reaction activity pursued by the vNSF, to result in more effective mitigation actions to security threats.

vNSF Application logic

It consists of one (or more) containers that run security and network This component reads application configuration from the Configuration volume and writes its output to the Log volume.

Instantiation work-flow

An overall depiction of the vNSF architecture is presented in Figure 3, along with the sequence of interactions between its components that shall occur during its instantiation.

1) The security policy is written to the Policy volume configured on the vNSF by a management entity, such as the NFV Virtual Network Function Manager;
2) the Policy translator reads the content of the Policy volume and processes the received data to produce application specific configuration rules;

3) the Policy translator writes the vNSF rules to the Configuration volume;
4) the vNSF Application logic is initialised with the configuration rules and starts to act according to its security control;
5) the vNSF Application logic reports the results of its security control to the Log volume;
6) the Log collector aggregates the vNSF logs and produces in output an evidence of the vNSF operation, to be further processed by the analytics.

In this work-flow, the vNSF is configured at instantiation time via the Policy translator, which can be removed afterwards. The current architecture does not provide any capabilities to update the state of the vNSF, and hence its security policies. This is due to the peculiarities of a container-based vNSF, which differentiate its behaviour from a traditional VM-based instance. First of all, best-practices in container development aim at stateless instances, which can be easily deployed in a micro-service environment. Moreover, the faster instantiation time and flexibility provided by OS-level virtualisation allows container management systems to quickly re-deploy instances in case their state is changed, instead of updating the running instances. The down-time due to the re-deployment can be mitigated by instantiating the updated vNSF before shutting down the obsolete instance, redirecting the traffic to the newest vNSF once it is fully loaded.

VI. Development of the prototype

In this section, we introduce the case study for the development of a vNSF prototype tailored for the SecaaS scenario. Then, we describe the technical aspects of the development process, focusing on the open-source technologies selected for each functionality.

Case study for the prototype

We focus on a practical case study for the development of a vNSF prototype, represented by an HTTP *Reverse Proxy* (RP) with embedded filtering capabilities via a *Web Application Firewall* (WAF).

A RP is an intermediate entity, or a *gateway*, that resides between a web client and one or more Origin Servers, whose role is to retrieve the web resources on behalf of such client. A RP has significant advantages from the infrastructural point of view, which becomes transparent to modification in the *back-end* network. It also allows load balancing of the incoming traffic, as the RP may adopt optimisation strategies to forward the client requests to different replicas of an Origin Server.

A RP also enables protection mechanisms on the incoming traffic, whose headers and payload may be analysed before forwarding the packets to the proper Origin Server. In this regard, a RP is often paired with a WAF, which employs a series of security controls based on known web-based attacks. A WAF can apply mitigations to web applications hosted on the Origin Servers with a centralised approach, without delegating the security checks on the web applications themselves. In this regard, a WAF can be used also to reduce the *Window of Exposure* to a new web-based attack, as all the back-end applications can be protected by a single update on the WAF itself.

The *Open Web Application Security Project* (OWASP) [16] and *The Web Application Security Consortium* (WASC) [17] both support *The Web Application Firewall Evaluation Criteria Project* (WAFEC), that aims to increase the use of Web Application Firewalls and also defines a series of criteria to help the security administrators to choose the best solution for their web applications [18]. WAFEC defines several architectures for the deployment of the WAF, i.e. *Reverse Proxy*, *Bridge*, *Router* and *Embedded* solutions.

The Reverse Proxy main advantage over the Bridge and Router alternatives is that it terminates the clients' sessions, such as encrypted TLS connections, allowing a deeper inspection of packets than transparent Layer 2-3 protection (up to application layer, i.e. Layer 7). Moreover, the Embedded solution, which implies the installation of a WAF on the application server itself, is not considered as a viable solution in the NFV SecaaS scenario, where an ISP may provide a vNSF to secure heterogeneous web application servers without additional development and integration costs by its clients.

Technology selection for the vNSF prototype

Different open-source technologies have been analysed to select the best RP and WAF combination. Regarding the RP software, we have considered *Apache HTTP Server* (HTTPD) [19], *Apache Traffic Server* [20], *Varnish Cache* [21], *HAProxy* [22] and *nginx* [23]. At the time of writing, HTTPD and nginx appear to be the most popular solutions [24] on the market. The starting point for the selection of the proper RP technology is represented by its integration with an open-source WAF. In this regard, *ModSecurity* [25] and *NAXSI* [26] WAF technologies have been taken into account, as the first can be integrated with both HTTPD and nginx, while the latter only interoperates with nginx. Therefore, we have evaluated different RP and WAF combinations, namely HTTPD with ModSecurity, nginx with ModSecurity and nginx with NAXSI.

The tests are based on a single machine, equipped with an Intel Core i7-4510U at 2 GHz, 12 GiB of DDR3 RAM, Debian 9 Stretch 64 bit Linux distribution with kernel Linux 4.9.0-3-amd64 and Docker version 17.06.0-ce. The different tests have been performed by running the evaluated software in a separate Docker container. We have used a benchmarking tool, i.e. *ApacheBench* (ab) [27] version 2.3, to simulate concurrent HTTP requests directed towards the test Origin Server.

Firstly, we have evaluated the number of requests satisfied by the Origin Server with and without a RP, to ensure that the Origin Server itself is not a bottleneck. In this regard, we have tested each of the RPs with concurrent HTTP requests performed by 1000 users in a timespan of 45 s. The result, depicted in Figure 4, shows that nginx has the worst impact on the total number of requests to be served by the Origin Server. Then, we have simulated HTTP connections with a *keep-alive* header, as suggested by WAFEC, for a timespan of 45 s and a number of 250, 500 and 1000 concurrent users. The

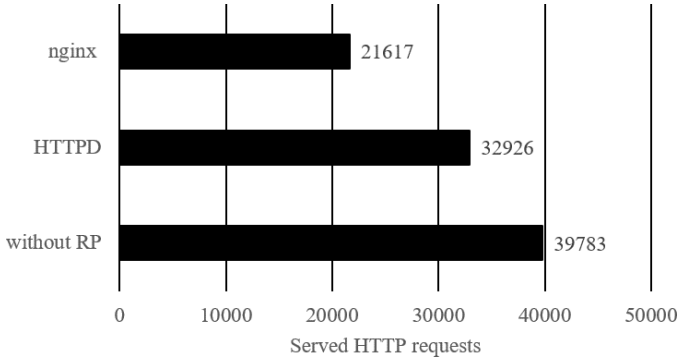

Fig. 4. Number of requests served by Reverse Proxies

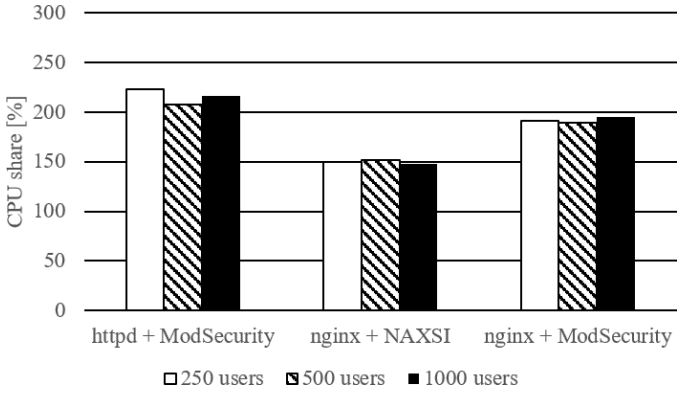

Fig. 5. CPU share by different RP and WAF combinations

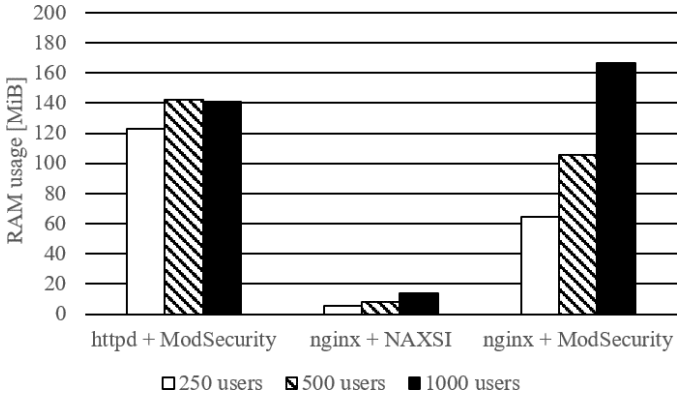

Fig. 6. RAM usage by different RP and WAF combinations

average CPU share by each of the tested solutions is reported in Figure 5. The different combinations have comparable performance, although the nginx-NAXSI solution minimises the CPU usage. Moreover, we have measured the average RAM usage of all the solutions, which result in a significant advantage of the nginx-NAXSI solution over the competitors, as depicted in Figure 6. Finally, we have measured the throughput of all of the RP and WAF combinations, by instantiating a single Origin Server in a Docker container, hosting a 32 KiB web page, and executing HTTP requests for 45 s. The result, depicted in Figure 7, shows that nginx-

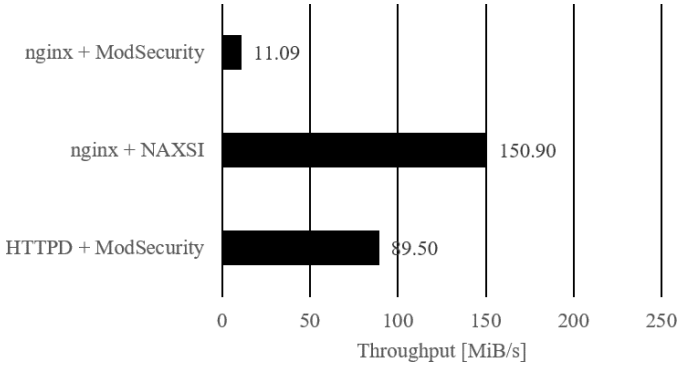

Fig. 7. Throughput of different RP and WAF combinations

NAXSI has the highest throughput, while nginx-ModSecurity is the worst performing solution. We have discarded the nginx-ModSecurity solution given its scarce experimental results. Although being promising from a performance perspective, the nginx-NAXSI solution has a significant disadvantage in terms of effectiveness in a NFV scenario. More specifically, NAXSI adopts a positive security model which evaluates the incoming traffic according to basic rules (e.g. the presence of certain characters, such as `<>, () {}`), computing a danger index for all of the packets. Only the packets that exceed a pre-defined threshold of the index are considered malicious traffic. This logic is effective to protect a web application from unknown vulnerabilities, but it requires a *learning* period that is not negligible in a NFV scenario, which requires a fast deployment and reaction to security threats.

ModSecurity adopts a negative, rule-based model that can be integrated with the OWASP rules for addressing common web application threats, such as *SQL Injection*, *Broken Authentication and Session Management* and *Cross-site Scripting* (XSS). Hence, the HTTPD-ModSecurity solution can be easily deployed on a network infrastructure and quickly provide WAF protection for the client' web services. Hence, the HTTPD-ModSecurity solution has been finally adopted for the vNSF prototype.

Work-flow manager

We have developed a component, named vNSF Controller, to act as a work-flow manager for the deployment of the vNSF. The main role of this component is to implement container-oriented life-cycle functionalities for the vNSFs, such as to instantiate and stop an instance, that are typically managed by the VNF Manager in a ETSI NFV MANO implementation. This application allows the reuse of the different vNSF Components among different vNSFs, more specifically the Policy translator and Log collector modules.

In order to instantiate a vNSF, we have defined a *manifest* file in JSON format to include all the parameters of the different vNSF Components and a description of their interconnection. The manifest includes identification data for the vNSF, information about each vNSF Component (as supported by the Docker engine semantics) and details about the volumes used

to interconnect the Docker containers. The vNSF Controller parses the manifest to initiate the deployment of the vNSF, which consists of the following steps:

1) build the Dockerfiles for each vNSF Component;
2) download the Docker images from the specified registry in the manifest;
3) create the volumes to interconnect the vNSF Components;
4) execute the vNSF Components.

The vNSF Controller internally executes the containers in a configurable order. In our architecture, the policy translator container must complete its execution before instantiating the vNSF application logic. The de-facto standard for defining multi-container Docker applications, named *Docker Compose*, does not ensure that a container has completed its execution before running the next instance [28]. Hence, we have developed a custom logic for managing the synchronisation in-between the execution of multiple containers in the vNSF Controller, leveraging the standard Docker API.

Policy translator

The policy specification language adopted for the prototype is the MSPL, as defined by the SECURED project, which is based on a XML language. We have developed a XSD schema tailored for the RP and WAF configuration, which abstracts the HTTPD and ModSecurity rules. Regarding the RP, we have included configuration rules for the different *virtual hosts* (e.g. domain names, Origin Server IP address and port). Regarding ModSecurity, the policy includes parameters to enable or disable standard WAF security controls and levels of inspection (i.e. headers and bodies) . An excerpt of a possible MSPL policy for the WAF is presented in Figure 8, which disables HTTP response body inspection, blocks SQL Injection attacks and logs requests performed by well-known black-listed User Agents.

```
<application-layer-condition>
  ...
  <request-body-inspection>
    ENABLED
  </request-body-inspection>
  <response-body-inspection>
    NOTENABLED
  </response-body-inspection>
  <pattern>
    <pattern-name>
      SQL_INJECTION
    </patter-name>
    <mode>ENABLED</mode>
  </pattern>
  <pattern>
    <pattern-name>
      BAD_ROBOTS
    </pattern-name>
    <mode>DETECTION_ONLY</mode>
  </pattern>
</application-layer-condition>
```

Fig. 8. Excerpt of an MSPL policy for the WAF vNSF.

Log collector

This component has been realised by leveraging the *Logstash* software, which is capable of receiving data from different sources, aggregate the information and forward them to a consumer, such as the DARE in the SecaaS scenario. The current implementation uses a volume to receive the log files in input, although a possible upgrade may leverage network-based log forwarding technologies, such as *syslog*, to eliminate the overhead in accessing the volume.

VII. Evaluation of the prototype

In this section, we present the experimental results of the evaluation of the vNSF prototype in different scenarios. First of all, we aim at demonstrating the low performance overhead introduced in the vNSF by the Docker encapsulation, against both a VM-based deployment and a bare-metal solution. In this regard, we adopt Linux KVM for the full virtualisation test, as it is widely utilised in cloud deployments and supported by the OpenStack compute node implementation. Regarding the Docker deployment, we utilise the Docker Engine installed on the host machine, in order to minimise the overhead introduced by virtualisation. The vNSF prototype is deployed on a host machine with a single *Network Interface Card* (NIC), which is connected to an isolated switch. Both the client user agent, represented by the ab benchmarking tool, and the Origin Server, a bare-metal deployment of an HTTPD server (hosting a single web page), are connected to the switch. The vNSF application logic leverages HTTPD 2.4.6 and ModSecurity 2.7.3-5.el7, and its host is equipped with a dual core CPU, the Intel Core i5-5300U at 2.3 GHz, 16 GiB of DDR3 RAM, running CentOS 7 64 bit with Linux kernel 3.10.0. The KVM scenario has been tested in a virtual instance equipped with a dual core virtual CPU and 1 GiB of RAM. The client machine is equipped with a dual core CPU, named Intel Core i7-4510U at 2 GHz, 12 GiB of DDR3 RAM, running Debian 9 Stretch 64 bit with Linux kernel 4.9.0-3-amd64 and ab 2.3. Finally, the Origin Server uses an Intel Core 2 6400 dual core CPU at 2.13 GHz, 2 GiB of DDR2 RAM, CentOS 7 64 bit with Linux kernel 3.10.0 and HTTPD 2.4.6.

The ModSecurity rule set adopted in all the tests is the default OWASP ModSecurity Core Rule Set enabled for all packet headers and request body only. The different tests described in this section have been performed on the same test-bed with comparable overall network usage, and the final results are computed as the average value of the measures recorded in three separate runs. The chosen metric is the throughput of the vNSF, expressed in MiBps, to demonstrate the low performance overhead introduced by containers against competitors. Another acceptable metric would be the introduced latency, and it will be considered for further

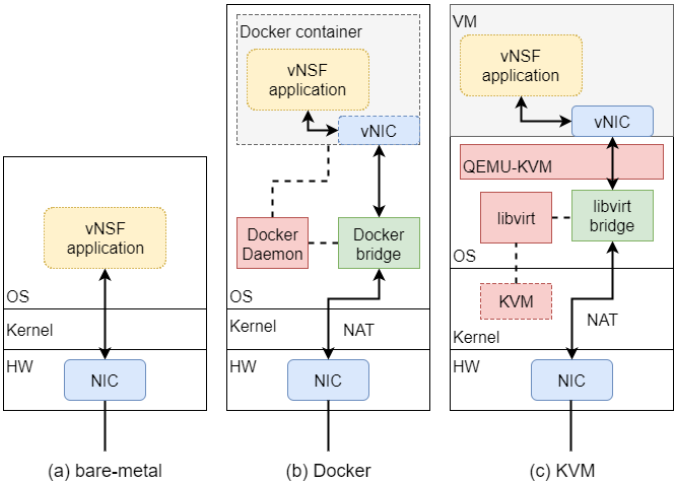

Fig. 9. vNSF deployment on a bare-metal server, Docker and KVM

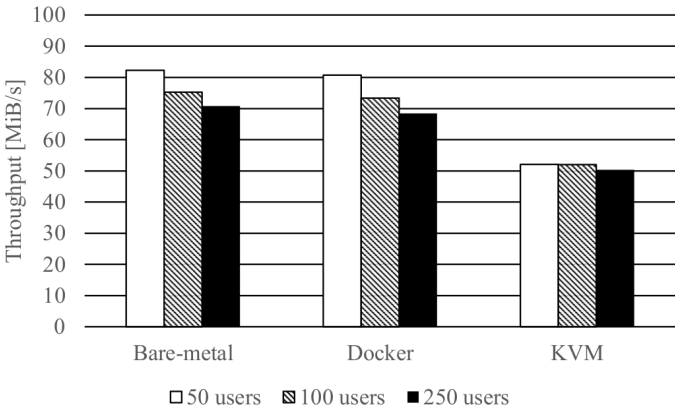

Fig. 10. vNSF throughput on bare-metal, Docker and KVM deployments

evaluation of the prototype.

Figure 9 depicts the different deployments of the vNSF, which include:

(a) bare-metal execution on the host machine;
(b) encapsulation in a Docker container, which is executed by the host machine Docker Engine and is interconnected to the NIC via the default Docker bridge;
(c) encapsulation in a KVM instance, which is executed via *libvirt* QEMU-KVM hypervisor driver [29] and is interconnected to the NIC via the default libvirt bridge.

Both case (b) and (c) leverage *iptables* [30], the packet filter of the Linux kernel, to provide *Network Address Translation* (NAT) based connectivity to the virtual instances, in order to let them reach the external network. In this scenario, we have performed a test by issuing a number of HTTP GET *keep-alive* requests for a timespan of 120 s, with a total of 50, 100 and 250 concurrent users. The Origin Server web page has a dimension of 40 KiB. The throughput expressed by each solution is plotted in Figure 10. The chart shows that the Docker-based deployment is comparable to the bare-metal solution, as its average throughput is 2% lower than the highest registered value. The KVM test has an average 32% lower throughput than the bare-metal scenario, justified by the overhead introduced by the full virtualisation network stack. We also note that the throughput decreases at the increase of concurrent users in each of the deployments. In the Docker scenario, the throughput registered with 250 concurrent users is 15% lower than the value registered with 50 clients.

We have performed a second test of the Docker-based vNSF in a load-balanced scenario, comprising the following components:

- the client user agent, i.e. the ab benchmarking tool;
- a host machine running the HAProxy load balancer, equipped with two different NICs;
- a switch that interconnects the load balancer with two application servers;
- two replicas of the container-based vNSF, hosted in identical application servers equipped with two NICs;
- a switch that interconnects the vNSF application servers with the Origin Server;
- an Origin Server, running a single web server as an HTTPD bare-metal deployment.

The links between the load balancer and the application servers have a 1 Gibitps bandwidth, while the other links have a 100 Mbitps bandwidth, due to avoid a load-balancing bottleneck. The machine hosting the Client user agent has the same hardware configuration of the previous test. The Origin Server is equipped with a dual core CPU, the Intel Core i5-5300U at 2.3 GHz, 16 GiB of DDR3 RAM, and it runs CentOS 7 64 bit with Linux kernel 3.10.0. Both the load balancer and the application server replicas are equipped with an Intel Core 2 6400 dual core CPU at 2.13 GHz, 2 GiB of DDR2 RAM, CentOS 7 64 bit with Linux kernel 3.10.0. The HAProxy load balancer is configured at TCP level and adopts a *round robin* policy for the forwarding of the requests to the replicas. In this scenario, we have evaluated the throughput of the vNSF without any replicas and no load balancer, by issuing a number of HTTP GET *keep-alive* requests in a timespan of 120 s with 50, 100 and 250 concurrent users. The Origin Server is configured to serve a single 40 KiB page. Then, we have performed the same test by including both the replicas and the HAProxy load balancer. The results are plotted in Figure 11, resulting in an average 71% increase of the throughput in the load-balanced scenario than the single vNSF deployment.

VIII. Future work

We aim at extending the research work described in this paper with respect to the overall architecture and its integration with upstream container management systems, in order to enhance maintainability and automation.

Regarding the architecture extension, we are focused on increasing the performance of the vNSF. This may be accomplished by investigating alternative methodologies to interconnect the vNSF Components rather than volumes, which introduce a significant overhead in terms of write and read access to the file system. Moreover, the log aggregation and pre-processing may be offloaded to an external, centralised

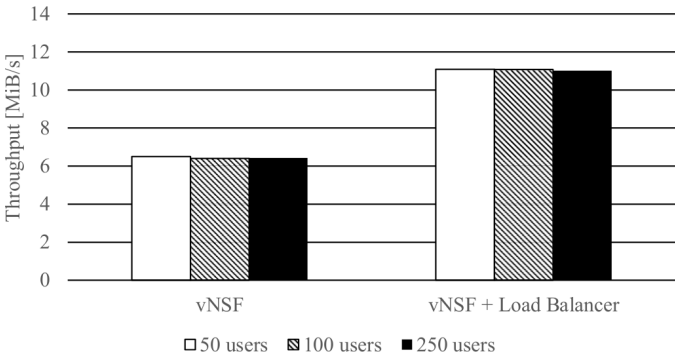

Fig. 11. Throughput on a load-balanced deployment with two vNSF replicas

entity of the cloud infrastructure, limiting the scope of the vNSF log collector to a distributed *log shipper*.

With respect to the integration with upstream container management systems, Kubernetes and Apache Mesos both provide the network and infrastructure automation in a multi-host container environment. In particular, Kubernetes is considered highly promising as it is already been integrated with the OPNFV [3] framework. In this regard, we are interested in both multi-host networking, to detach the vNSF Components from a single host deployment, and the automation of the vNSF instantiation (that we have proved in the vNSF Controller PoC). Regarding the multi-host networking, we plan to investigate advanced Docker network technologies, such as the built-in *overlay* driver and the third-party Weave Net solution. The Open Source MANO VIM emulator may be an alternative target platform for the integration of the proposed solution, although it currently limits the deployment of containers to a single node.

IX. Conclusion

In this research work, we investigate the peculiarities of developing a container-based vNSF, which results in the design of a multi-container architecture to include vNSF Components that suit the policy translation and log reporting capabilities of the ETSI NFV SecaaS scenario. This architecture is different from a VM-based vNSF with respect to the *intra-vNSF* interactions and segmentation of duties in different virtual instances, according to the *one process per container* paradigm and for the sake of modularity. Moreover, in this paper we propose a concrete implementation of such architecture for a Reverse Proxy vNSF, enhanced with filtering capabilities of a Web Application Firewall. In this regard, we have compared several open-source technologies, evaluating their performance and suitability to the NFV scenario. In this research, we partially address the management and orchestration requirements of a vNSF, by proposing a PoC for a standalone vNSF Controller that could be further integrated with a container management system. Finally, the experimental evaluation of our prototype shows that the proposed solution outperforms a VM-based vNSF with the same application logic, and can be also applied in a load-balanced scenario for improved resilience and performance.

Acknowledgment

The research described in this paper is part of the SHIELD project, co-funded by the European Commission (H2020 grant agreement no. 700199). We thank Vincenzo Paolo Bacco (vincenzopaolobacco@gmail.com) for his significant contribution to the design and development of the vNSF prototype.

References

[1] Open Baton. [Online]. Available: https://openbaton.github.io/
[2] Docker. [Online]. Available: https://www.docker.com/
[3] OPNFV. [Online]. Available: https://www.opnfv.org/
[4] Open Source MANO. [Online]. Available: https://osm.etsi.org/
[5] VIM emulator. [Online]. Available: https://osm.etsi.org/wikipub/index.php/VIM_emulator
[6] R. Cziva, S. Jouet, K. J. S. White, and D. P. Pezaros, "Container-based network function virtualization for software-defined networks," in *IEEE Symposium on Computers and Communication (ISCC)*, Larnaca (Cyprus), July 6-9, 2015, pp. 415–420.
[7] R. Cziva and D. P. Pezaros, "Container network functions: bringing nfv to the network edge," *IEEE Communications Magazine*, vol. 55, no. 6, pp. 24–31, 2017.
[8] J. Anderson, H. Hu, U. Agarwal, C. Lowery, H. Li, and A. Apon, "Performance considerations of network functions virtualization using containers," in *International Conference on Computing, Networking and Communications (ICNC)*, Kauai, Hawaii (USA), February 15-18, 2016, pp. 1–7.
[9] "ETSI GR NFV-EVE 004 Network Functions Virtualisation (NFV); Virtualisation Technologies; Report on the application of Different Virtualisation Technologies in the NFV Framework ," ETSI NFV ISG, Mar. 2016.
[10] "ETSI GS NFV-INF 004 Network Functions Virtualisation (NFV); Infrastructure; Hypervisor Domain," ETSI NFV ISG, Jan. 2015.
[11] "ETSI GS NFV-SWA 001 Network Functions Virtualisation (NFV); Virtual Network Functions Architecture," ETSI NFV ISG, Dec. 2014.
[12] "ETSI GS NFV-SWA 001 Network Functions Virtualisation (NFV); Use Cases," ETSI NFV ISG, May 2017.
[13] "ETSI GS NFV-SEC 013 Network Functions Virtualisation (NFV) Release 3; Security; Security Management and Monitoring specification," ETSI NFV ISG, Feb. 2017.
[14] SECURED. [Online]. Available: https://www.secured-fp7.eu/
[15] SECURED - D4.1 Policy specification. [Online]. Available: https://www.secured-fp7.eu/files/secured_d41_policy_spec_v0100.pdf
[16] OWASP. [Online]. Available: https://www.owasp.org/index.php/Main_Page
[17] WASC. [Online]. Available: http://www.webappsec.org/
[18] "Web application firewall evaluation criteria," Web Application Security Consortium, Jan. 2006. [Online]. Available: http://projects.webappsec.org/f/wasc-wafec-v1.0.pdf
[19] Apache HTTP Server. [Online]. Available: https://httpd.apache.org/
[20] Apache Traffic Server. [Online]. Available: http://trafficserver.apache.org/
[21] Varnish HTTP Cache. [Online]. Available: https://varnish-cache.org/
[22] HAProxy. [Online]. Available: http://www.haproxy.org/
[23] nginx. [Online]. Available: https://www.nginx.com/
[24] "June 2017 web server survey," Netcraft. [Online]. Available: https://news.netcraft.com/archives/2017/06/27/june-2017-web-server-survey.html
[25] ModSecurity. [Online]. Available: https://modsecurity.org/
[26] NAXSI, an Nginx Web Application Firewall. [Online]. Available: https://www.nbs-system.com/en/it-security/it-security-tools-open-source/naxsi/
[27] ab - Apache HTTP server benchmarking tool. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html
[28] Docker - Control startup order in Compose. [Online]. Available: https://docs.docker.com/compose/startup-order/
[29] Libvirt - KVM/QEMU hypervisor driver. [Online]. Available: https://libvirt.org/drvqemu.html
[30] The netfilter.org iptables project. [Online]. Available: https://www.netfilter.org/projects/iptables/index.html