2016 5th IEEE International Conference on Cloud Networking

Enabling NFV Services on Resource-Constrained CPEs

Roberto Bonafiglia*, Sebastiano Miano*, Sergio Nuccio†, Fulvio Risso*, Amedeo Sapio*
*Politecnico di Torino, Department of Computer and Control Engineering, Torino, Italy
†Telecom Italia, Torino, Italy
*{roberto.bonafiglia, sebastiano.miano, fulvio.risso, amedeo.sapio}@polito.it †sergio.nuccio@telecomitalia.it

***Abstract*—Virtual Network Functions (VNFs) are often implemented using virtual machines (VMs) because they provide an isolated environment compatible with classical cloud computing technologies. Unfortunately, VMs are demanding in terms of required resources and therefore are not suitable for resource-constrained devices such as residential CPEs. However, such hardware often runs a Linux-based operating system that supports several software modules (e.g., iptables) that can be used to implement network functions (e.g., a firewall), which can be exploited to provide some of the services offered by simple VNFs, but with reduced overhead. In this paper we propose and validate an architecture that integrates those native software components in a Network Function Virtualization (NFV) platform, making their use transparent from the user's point of view.**

***Keywords*-Customer Premise Equipment; Network Function Virtualization; Virtual Network Function; Home Networks; Home Gateway; Home Security;**

I. Introduction

While cloud providers can count on centralized data centers encompassing mainly homogeneous servers, telecom operators feature an existing widely distributed infrastructure made of heterogeneous devices. In particular, although we can see clear benefits by integrating current Customer Premise Equipment (CPE) in the Network Functions Virtualization (NFV) infrastructure [1], those devices are usually based on low-cost hardware that cannot support Virtualized Network Functions (VNFs) under the form of virtual machines.

However, we can note that most CPEs are based on the Linux operating system, which includes (hence it can potentially execute) a broad set of existing software network functions (e.g., firewall, NAT, virtual switch, etc) running on the bare hardware.

This paper exploits this capability and proposes a software architecture that integrates existing CPEs in an NFV domain, leaving complex VNFs in the data center while simple *Native Network Functions (NNFs)* are executed in the CPE with low hardware resources, especially on the Home Gateway, hence combining the benefits of the cloud with the locality of the services running on local CPEs.

NNFs rely on the *native capabilities*, i.e. software components that are already available on the CPE and that can be executed directly on the host operating system. In particular, the concept of "native" involves not only regular and built-in network functions (such as a virtual switching instance), but also elements (e.g., the Linux *iptables* module) that can be exploited to build network services (e.g., a firewall). As a result, native functions lead to significant improvements, in terms of memory consumption, storage requirements and start-up time, compared to existing technologies (LXC, Docker, VMs), enabling the execution of network functions even on resource-constrained devices.

Our solution enables an NFV orchestrator to optimize the scheduling of the Network Functions (NFs) by starting the services that require network functions close to the end user (e.g., IPsec terminator, low-latency services) directly on the user CPE, while other components of the same service (e.g., the NAT module) are executed in a remote data center. This requires our architecture to define an abstraction that allows the orchestrator to understand the capabilities of the underlying infrastructure domain, and that can handle the lifecycle of each network function independently from its actual implementation. Furthermore, a reasonable security model has to be defined in order to support multi-tenancy for NNF as well, as the nice properties in terms of security and isolation guaranteed by traditional hypervisors are not available in our context.

The rest of this paper is organized as follows. The next section examines similar works in current research. Section III describes the technologies used in this work. Section IV presents and describes Native Network Functions with their abstraction. Experimental results that validate this work are shown in Section V, followed by some final considerations and conclusions in Section VI.

II. Related Work

The necessity to introduce more flexibility in CPEs serving home/small office customers has increased over the years and has become evident with the emerging NFV paradigm. In fact, a recent trend consists in moving (part of) the CPE functions in the data center with the so called virtual CPE (vCPE) such as in [2], [3]; a minimal hardware appliance is left at the edge of the network, while (most of) the intelligence is moved to the cloud and implemented through virtual functions. An intermediate step toward a fully virtualized CPE is proposed in [4], which is based on the architecture defined by the Home Gateway Initiative industry alliance[1]. This architecture

[1]http://www.homegatewayinitiative.org/. However, this working group will be shut down in 2016.

978-1-5090-5093-2/16 $31.00 © 2016 IEEE
DOI 10.1109/CloudNet.2016.24

is highly modular and implements the different CPE functions as Java OSGi bundles, which can be dynamically loaded/discarded on demand. The *Surrogate vNF* proposed by the paper extends this paradigm by defining a set of "proxy" OSGi functions that keep compatibility with the existing architecture while delegate most of the processing to a companion VNF running in the cloud.

However, the above solutions require excellent connectivity between the customer premises and the data center, and may introduce excessive delay for some latency-sensitive services. Furthermore, although in principle NFV enables a telecom operator to orchestrate its services by exploiting the resources offered across its entire network infrastructure, the vCPE approach cannot exploit resources that may be available on the CPE itself as VNFs are moved to the cloud.

Edge-based services are proposed in [5], which exploits eBPF programs to create a programmable data path in the CPE while the control plane is kept on the cloud. The CPE is able to handle locally the traffic, hence guaranteeing its operations also in case the connectivity toward the cloud is lost. Although this solution is very efficient, the eBPF virtual machine is not Turing-complete and cannot support even simple programs (e.g., string matching) that are rather common at the edge of the network.

Considering that the CPE is usually resource-constrained, [6] proposes an optimization model that is able to select the best VNF among a set of possible choices, hence optimizing the cost of the VNFs deployed on CPE. However, they rely on the existing technologies for the VNF implementation such as Linux containers or virtual machines, thus being orthogonal with the idea proposed in this paper.

To summarize, current NFV-compatible solutions do not support local processing in the CPEs, while more flexible CPE architectures are still limited in terms of supported features and are not compliant with the NFV world. This paper aims at achieving both objectives, namely NFV compatibility and arbitrary traffic processing in the CPE, while still supporting possible VNF running in the cloud, if needed.

III. Background

The architecture proposed in this paper, depicted in Figure 2, is an extension of the *Universal Node* (UN) [7] developed in the EU UNIFY project [1].

The UN is a single box, e.g., a server, that features a control plane that is able to jointly orchestrate network and compute resources, supports multiple execution environments and different virtual switches, and advertises functional capabilities (e.g., capability to execute a NAT service) instead of infrastructure-like information (e.g., KVM execution environment, available memory, etc.). The UN is a tiny infrastructure domain and it exploits locally available information to optimize the service evaluating local resources/constraints, such as assigning VNFs to the best CPU cores.

Upon accepting a new service request from an overarching orchestrator that is in charge of the global deployment of the service across multiple infrastructure domains, the UN

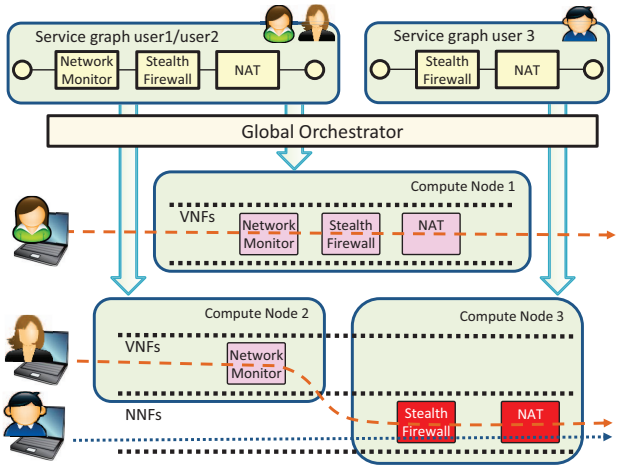

Figure 1. Service instantiation of a graph.

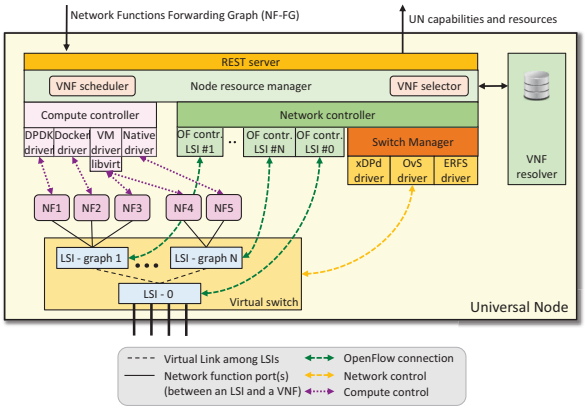

Figure 2. Architecture of the Universal Node.

can either deploy exactly the VNFs requested by the global orchestrator or, in case "generic" VNFs are chosen (e.g., a generic firewall instead of the one of a specific manufacturer) it relies on an additional component, the VNF resolver, to select the best implementation available. Furthermore, it creates a new Logical Switching Instance (LSI) to properly steer the traffic among the selected VNFs.

A. Network abstraction

The network controller of the UN manages the networking paths among the deployed VNFs through multiple levels of LSIs: a base LSI-0 and a set of LSI-N (where $N \geq 1$), each one in charge of a different deployed graph (Figure 1). The first (LSI-0) dispatches the traffic from the physical interfaces of the machine to the LSIs of the other graphs. The additional LSIs create the traffic steering paths between the VNFs that belong to that graph. Each LSI is managed by a separate embedded OpenFlow controller, provided by the UN, that dynamically inserts the proper rules in its flow table(s).

A switch manager module can control different types of virtual switches by means of the set of primitives listed in

Table I
NETWORK ABSTRACTION IN THE UN

Function	Description
createLSI()	Create an LSI
deleteLSI()	Remove an LSI
createPort()	Create a port connected to a NF on an LSI
deletePort()	Remove a port connected to a NF from an LSI
createTSRule()	Generate a new traffic steering rule in an LSI
deleteTSRule()	Remove an existing traffic steering rule from an LSI

Table II
COMPUTE ABSTRACTION IN THE UN

Function	Description
createNF()	Allocate the resources to start the NF; create a shadow (local) copy of the NF image (if needed)
startNF()	Attach ports to the NF, and starts the NF image
stopNF()	Stop the NF, without deallocating resources
updateNF()	Update the NF while running, e.g., by removing or hotplugging new network interfaces
deleteNF()	Release the resources (memory, shadow disk image) allocated to the NF
pauseNF()	Suspend the NF execution (for possible migration)

Table I and implemented by each technology-specific driver. Basically, the abstraction allows to *(i)* create/delete an LSI, *(ii)* create/remove a port on the LSI that is connected to a NF, and *(iii)* create/remove a traffic steering rule between VNFs or ports. This allow to replace a generic virtual switch implementation with an hardware-accelerated one, without impacting on the rest of the software.

B. Compute abstraction

The compute controller is responsible for the VNF lifecycle management, such as instantiate, terminate and update a VNF. This is achieved by defining a common compute abstraction (Table II) that is generic enough to be applicable to any type of execution environment. This abstraction is implemented by a set of drivers, each one in charge of a specific execution environment technology (e.g., VM, Docker, DPDK process) with the associated required parameters. For instance, the plugin that manages the KVM hypervisor creates on the fly the proper XML file required by the `libvirt` library for the VM instantiation when the `createNF()` call is invoked.

Each compute driver needs also to support different types of interfaces (e.g., `dpdkr`, `virtio`, etc.), according to the specific execution environment, as each execution environment supports only a subset of the available port types. In this respect, the compute controller needs to be coordinated with the network controller in order to attach the VNF ports, according to the required technology, to the existing softswitch.

C. Northbound interface

The northbound interface of the UN is bidirectional: the downstream direction is based on generic service graphs that have to be instantiated on the node, while the upstream direction is used to export the information needed by an overarching orchestrator and that is used to properly instantiate the requested service across different infrastructure domains.

The UN exports three types of information to such an upper layer orchestrator. *Functional capabilities* represent the ability to execute a specific network function, such as a NAT or firewall service, optionally with some specific characteristics, such as the capability to handle high amount of traffic (e.g., because it can exploit an hardware accelerator available in the node). *Infrastructure-level capabilities* refers lower-level characteristics, such as the CPU architecture, the ability to execute generic VMs or Docker containers, etc. *Available resources* refer to the about of unused hardware resources, such as the amount of free memory or the presence of an hardware accelerator.

The capability to advertise functional capabilities is a unique feature of the UN and it represents also the main reason we can bring the concept of Native Network Functions in this environment. In fact, an overarching orchestrator that operates based on functional capabilities will not decide the actual VNF implementation to be used, but it will only tell the underlying domain (e.g., the UN) to start a specific network function, leaving to that domain the decision about the specific VNF flavor (e.g., VM, Docker, etc) to be used. In turn, the UN will delegate this decision to the the VNF resolver.

IV. NATIVE NETWORK FUNCTIONS

This section introduces the concept of Native Network Function, i.e. a data-plane processing component that exploits capabilities natively present on the compute node and cannot be exploited by current NFV solutions. Our architecture allows NNFs to work alongside traditional VNFs, giving the possibility to improve overall network performance without losing the flexibility guaranteed by the NFV approach.

A. NNF model and VNF template

In principle, a generic NNF must be compliant with the interface defined for all the compute drivers, such as in Table II. However, each NNF may require some additional properties that have to be satisfied in order to be able to run. Therefore, besides all the information required for the execution of a generic VNF (e.g., number of ports, port types), additional properties refer to either *dependencies* or *requirements*. Those might refer to software packages (e.g., executables, libraries) available on the compute node that are already installed and that are required for the NNF to operate.

In addition, our model considers also information regarding the *status* of the allocated function, telling about current configuration and resource used by the function. This data is needed in order to be able to release the resources used by the NNF when the function stops, while in traditional VMs resources are freed along with the deletion of the VM.

In order to cope with this data, we extended the *network function template* to keep both VNF general attributes, common for all types of network function, and NNF-specific

```
{
     "name" : "firewall" ,
     "uri" : "http://repo/native/firewall.tgz",
     "vnf-type" : "native",
     "multitenancy" : true,
     "dependencies" :
     { "capability": [ {
          "name" : "iptables",
          "type" : "package" ,
          } ] },
}
```

Figure 3. Excerpt of the template of a firewall NNF.

information. Figure 3 contains an excerpt of an NNF template representing a native firewall, which shows the properties of the function. In particular, it exploits *iptables* as a native capability and also supports multi-tenancy. The function handlers that will be used by the compute controller to drive its lifecycle (e.g., start, modify and stop) are available at the given URI with a specific format. The template also specifies basic I/O and network configuration of the function, information needed for driving the other NF types as well, not shown in the example.

B. The native compute driver

After receiving the VNF template, the compute controller has to control the native function by using the abstraction described in Section III-B. The native driver will download the function using the URI specified in the VNF template, which points to a `.tgz` file. The above archive is a very compact file that includes a set of bash scripts that are called to perform the actions listed in Table II, such as starting a new instance of the NNF, updating, stopping and all the other actions that are required in the VNF lifecycle management. As evident, the support for bash is the only requirement for running a NNF, which, in turn, enables native functions to be seamlessly deployed on machines with different CPU architectures.

C. I/O model

In the traditional NFV framework, the traffic steering among the VNFs is carried out by a virtual switch that forwards packets according to the rules given by a network controller. Each VNF is provided by a certain number of virtual network interfaces that correspond to its ports, connected to the virtual switch.

In order to seamlessly support the execution of NNFs, the same I/O model must be repeated and therefore each NNF should be connected to the vSwitch with the appropriate number of ports. In this way, the network controller remains exactly the same and can create virtual ports for the NNF as well as for the VNF.

In the NNF case, these ports are implemented as virtual Ethernet (veth) interfaces assigned to a network namespace on which the NNF is executed. As such, each NNF sees its own network interfaces that can use to retrieve/send its own specific network traffic.

D. Isolation model

Differently from current virtualization technologies that natively support an isolation model for the instantiated VNFs, the NNF driver needs to explicitly implement a layer that provides at least some form of isolation of the NNF against the rest of the system.

The NNF driver leverages the Linux namespaces by creating a new *network* namespace before running an NNF, adds to it the virtual network ports required by the function, and then launches the NNF inside the namespace. As a result, the NNF is isolated for the incoming traffic that will be only the one sent by the vswitch to the veth of the NNF. The name of the namespace is unequivocally related with the graph and the function name, thus avoiding possible collisions. At the end of the execution of the NNF, the namespace is deleted by the native driver and all the other related resources are freed.

Differently from Linux containers that exploit *all* the different types of namespaces available in the Linux OS, NNFs use by default only the network one in order to guarantee network isolation between different NNFs. A more sophisticated isolation model, leveraging multiple namespaces that can be activated on demand (based on the requirements of the tenant, the infrastructure owner, and NF), is currently in progress.

E. Multitenancy

In a traditional NFV architecture in which each VNF runs on a distinct VM, multitenancy is an intrinsic property of the execution model. In fact, multiple instances of the same VNF can always be launched while traffic steering primitives can set the proper flow rules to the software switches in order to create the correct traffic steering paths among VNFs.

Supposing that a NNF can be instantiated multiple times, multitenancy is achieved by encapsulating multiple instances of the NNF in dedicated namespaces whose virtual interfaces are connected to different ports of the software switches. On the contrary, if a NNF does not support multiple instances running at the same time, multitenancy should be managed by means of an ad-hoc marking mechanism that allows the NNF to distinguish between traffic belonging to different service graphs.

F. Security considerations

Launching a native function, hence a script running on the bare hardware, offers less protection than starting a software in a VM or in a Docker container, which can leverage the additional protection shield provided by the hypervisor or the Docker execution engine. For instance, little protection exists to limit the resources used by native functions, e.g., in terms of CPU/memory consumption or the number of occupied CPU cores. Although the impact of the above problems could be limited by turning on some addition Linux mechanism such as `cgroup`, this complicates the solution to the extent to which other alternatives may be more appealing, such as replacing the NNF with a Docker-based implementation.

In any case no protection exists that prevents a VNF, which is expected to provide a given service (e.g., firewall), to behave

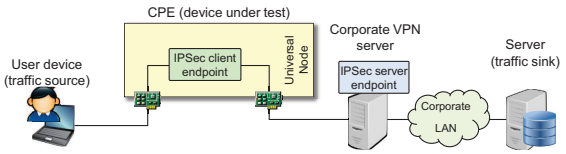

Figure 4. Testbed used in the validation.

Table III
CHARACTERISTICS OF THE DEVICES USED IN THE VALIDATION

Machine(s)	Hardware and software characteristics
User device (source) Traffic server (sink) Corp. VPN server	Intel Core i7-4770, 32GB RAM, 500GB HD Linux Ubuntu 14.04, Kernel version: 3.16
Server CPE	Intel Core i5-3450S, 8GB RAM, 200GB SSD Linux Ubuntu 14.04, Kernel version: 3.19
Domestic CPE	Netgear R6300v2, CPU Broadcom BCM4708A0, 800MHz (2 cores), 128MB Flash, 256MB RAM OpenWrt 15.05, Kernel version: 3.18
Business CPE	Hawkeye HK-0910, Freescale QorIQ T1040, 1.2GHz (four e5500 cores), 64MB NOR Flash, 2GB RAM DDR3L-1600 Freescale QorIQ SDK V1.7, Kernel version: 3.12

differently (e.g., to launch an attack toward a remote host) and the current solution is simply to trust the creator of the application or the entity (e.g., app marketplace owner) that sells it. Therefore, although we acknowledge that the problem of determining whether a NF is malicious is emphasized in case of NNF because of their inferior degree of isolation, we feel that the problem is rather general and should require a more generic solution that guarantees, a priori, the goodness of the VNF, e.g., by means of novel software verification techniques.

In this respect, a possible direction for future investigation could consist in integrating remote attestation techniques [8] in our execution environment, exploiting an external machine to verify the correctness of the running software.

V. VALIDATION

This section presents the results of a preliminary validation campaign with a transparent VPN access use case, i.e., when a user client located on a trusted local network (e.g., home) needs to connect to its corporate VPN server. In order to avoid the necessity to install the VPN client software on all user's devices (e.g., laptop, smartphone, etc.), the VPN client is instantiated on the user's CPE, hence providing secure access to the corporate network independently of the user device.

The specific testbed, shown in Figure 4, encompasses a client that generates the traffic, a CPE executing the IPSec client NF in charge of encrypting/decrypting the traffic, a VPN server with the opposite duty, and finally a traffic sink. All the four boxes are connected with point-to-point 1Gbps Ethernet links; faster speeds are usually not available in low-end CPEs. Three powerful workstations were used respectively as traffic source, VPN server and traffic sink in order to avoid those machines to become the bottleneck, while different flavors of CPEs are used, namely a mid-range server, a business CPE based on the Freescale T1040 and a domestic CPE, all with the same version of the UN software, although compiled for the specific platform. The specific hardware and software details are listed in Table III.

The UN was configured through its northbound interface with a very simple service graph, featuring an IPsec client NF connected to the LAN and WAN ports; the NF was based on the well-known Strongswan [9] software, configured to operate in IPsec tunnel mode (using IKEv2 to establish the security associations, AES-CBC-128 for the encryption and SHA1-HMAC for verifying the data integrity).

The use of different hardware platforms was coupled with different implementations of the same NF, whenever possible. The server-based CPE was tested with three equivalent network functions based on VM, Docker and NNF, while the business CPE and the domestic CPE supported the network function only as sofware-based NNF. Our experiments took into consideration (*i*) the throughput between the two hosts and the associated CPU load during the experiment, (*ii*) the amount of RAM consumed, (*iii*) the NF image size, (*iv*) the amount of additional libraries required to start the requested execution environment in addition to the base Linux system (e.g., KVM/QEM for VMs) and (*v*) the time required to start the NF. The first two experiments leveraged the `iperf` tool installed on the source and sink machines, configured to generate two unidirectional TCP streams at the maximum speed. We set the packet size such that the MTU is not exceeded after the addition of the IPSec header, in order to avoid fragmentation. All experiments were repeated 10 times and averaged.

The *throughput*, in the second column of Table IV, shows that NNFs and Docker bring significant performance improvements compared to VMs because of the simplified architecture that does require neither the hypervisor nor the guest OS, where the NF is running. Their throughput is higher with a reduced CPU consumption as well. In this respect, NNFs and Docker show the same level of performance, as expected, given that they are based on the same technology (i.e., kernel-based processing in the host plus namespaces).

The *memory occupation*, i.e., the amount of RAM required to execute the given NF and the execution platform, showed in the third column of Table IV, exhibits the same trend. In this case numbers can only be considered as qualitative measurements, as they may change considerably by tuning the NF in a different way, particularly for the VM case. In our test we created a guest OS with the default installation of a Ubuntu server 14.04, installing only the packages required for our VNF to work. As evident, the memory occupation is definitely higher in the case of VMs, while Docker and NNF are very similar, although they slightly vary according to the hardware platform under consideration. Note that Table IV reports the application-level throughput, i.e., measured on the source/sink machines. Packets are extended with the additional

Table IV
COMPARING DIFFERENT IMPLEMENTATIONS OF THE IPSEC CLIENT, ON DIFFERENT MACHINES

IPsec client implementation	Thr.@CPU (Mbps/load%)	RAM (MB)	NF image (MB)
1) Server CPE - KVM/QEMU	796 / 100%	390.6	522
2) Server CPE - Docker	1095 / 80%	24.2	240
3) Server CPE - NNF	1094 / 80%	19.4	5
4) Domestic CPE - NNF	57.2 / 100%	5	2
5) Business CPE - NNF	617 / 90%	1.9	3.7

IPsec headers required to create the tunnel, hence reaching, between the CPE and the IPSec server endpoint, an higher throughput.

The fourth column of Table IV shows the *NF image size*, which confirms definitely the advantages of NNFs not only with respect to VMs, but also against Docker, as the image size is about two orders of magnitude less than its counterparts[2]. Moreover, this impacts also on the time required to download the NF image from a remote location, which is critical when the CPE is connected to the Internet through slow links (e.g., ADSL). An additional test was carried out in the host environment to measure the *additional disk size*, required in the host, to support the execution of the specific environment; due to the intrinsic limitations this was only possible on the server-based CPE. Starting from a clean installation of Ubuntu server 14.04 with default settings, we measured an additional 40MB for the components (i.e., KVM/QEMU) required to execute VMs and 30MB required to execute Docker containers. The above numbers confirm the advantages of the NNF with resource-constrained environments; in fact, the reason for not testing Docker on the home and business CPEs is the disk size limitation on those platforms.

Finally, we measured also the *time to start a NF* in the server-based CPE, being the only environment that can start all the NF types. The result showed about 3 second with VMs (which require starting the entire VM), 350ms with Docker, and 727ms with NNF; the baseline, i.e., the time required to launch the IPsec client on the base system without wrapping it in any NF, was 154 ms. This confirms, once more, the advantage of running applications in the host; the (relatively) high number with NNF is due to some implementation-dependent delay required to attach the network ports to the NNF, requires further optimizations.

VI. CONCLUSIONS

This paper presents the idea of *Native Network Functions*, an NFV abstraction that allows to execute network functions even on resource-constrained devices by exploiting their native (both software and hardware) capabilities.

Our preliminary validation campaign confirms that NNFs can be implemented over a reasonable variety of hardware, ranging from standard high-volume servers to business and domestic CPEs, with different hardware characteristics (CPU architecture and speed, memory size, etc.). Furthermore, NNFs can export existing hardware accelerators as network functions, hence enabling an NFV orchestrator to transparently take advantage from the superior efficiency of the hardware compared to pure software implementations.

Future work will aim at extending this approach to support traditional middleboxes as well (e.g., routers, switches, etc.), allowing their seamless integration in an existing NFV infrastructure.

[2]The image size of a NNF is merely the size of the NF software, compiled for the target platform

ACKNOWLEDGMENT

This work was conducted within the framework of the FP7 UNIFY[3] and SECURED[4] projects, which are partially funded by the Commission of the European Union. Study sponsors had no role in writing this report. The views expressed do not necessarily represent the views of the authors' employers, the UNIFY and SECURED projects, or the Commission of the European Union.

[3]http://www.fp7-unify.eu/

[4]http://www.secured-fp7.eu/

REFERENCES

[1] A. Császár, W. John, M. Kind, C. Meirosu, G. Pongrácz, D. Staessens, A. Takács, and F.-J. Westphal, "Unifying cloud and carrier network: Eu fp7 project unify," in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, IEEE. Washington, DC, USA: IEEE Computer Society, December 2013, pp. 452–457.

[2] Z. Bronstein and E. Shraga, "Nfv virtualisation of the home environment," in *Consumer Communications and Networking Conference (CCNC), 2014 IEEE 11th*, Jan 2014, pp. 899–904.

[3] T. Cruz, P. Simes, N. Reis, E. Monteiro, and F. Bastos, "An architecture for virtualized home gateways," in *Integrated Network Management (IM 2013), 2013 IFIP/IEEE International Symposium on*, May 2013, pp. 520 – 526.

[4] N. Herbaut, D. Negru, G. Xilouris, and Y. Chen, "Migrating to a nfv-based home gateway: Introducing a surrogate vnf approach," in *Network of the Future (NOF), 2015 6th International Conference on the*, September 2015, pp. 1–7.

[5] F. Sanchez and D. Brazewell, "Tethered linux cpe for ip service delivery," in *Network Softwarization (NetSoft), 2015 1st IEEE Conference on*. IEEE, April 2015, pp. 1–9.

[6] G. Faraci and G. Schembra, "An analytical model to design and manage a green sdn/nfv cpe node," *Network and Service Management, IEEE Transactions on*, vol. 12, no. 3, pp. 435–450, September 2015.

[7] I. Cerrato, A. Palesandro, F. Risso, M. Suñé, V. Vercellone, and H. Woesner, "Toward dynamic virtualized network services in telecom operator networks," *Computer Networks*, vol. 92, Part 2, pp. 380 – 395, 2015, software Defined Networks and Virtualization.

[8] T. Su, A. Lioy, and N. Barresi, "Trusted computing technology and proposals for resolving cloud computing security problems," in *Cloud Computing Security: Foundations and Challenges*, In Press.

[9] "Strongswan," https://www.strongswan.org/, [Online; accessed August 07th, 2016].