FOP4: Function Offloading Prototyping in Heterogeneous and Programmable Network Scenarios

Daniele Moro*, Manuel Peuster†, Holger Karl†, and Antonio Capone*
*Politecnico di Milano: {daniele.moro, antonio.capone}@polimi.it
†Paderborn University: {manuel.peuster, holger.karl}@upb.de

***Abstract*—Offloading packet processing tasks to programmable switches and/or to programmable network interfaces, so called "SmartNICs", is one of the key concepts to prepare softwarized networks for the high traffic demands of the future. However, implementing network functions that make use of those offloading technologies is still challenging and usually requires the availability of specialized hardware. It becomes even harder if heterogeneous services, making use of different offloading and network virtualization technologies, should be developed.**

In this paper, we introduce *FOP4* (Function Offloading Prototyping with P4), a novel prototyping platform that allows to prototype heterogeneous software network scenarios, including container-based, P4-switch-based, and SmartNIC-based network functions. The presented work substantially extends our existing Containernet platform with the means to prototype offloading scenarios. Besides presenting the platform's system design, we evaluate its scalability and show that it can run scenarios with more than 64 P4 switch or SmartNIC nodes on a single laptop. Finally, we presented a case study in which we use the presented platform to prototype an extended in-band network telemetry use case.

I. Introduction

Concepts like network function virtualization (NFV) and software defined networking (SDN) are the main enablers for agility and manageability in future networks. However, the packet processing performance of software entities running, e.g., in a virtual machine (VM) or container, so-called virtualized network functions (VNF), are often not sufficient to effectively cope with increasing traffic demands. To this end, the concept of *offloading* packet processing tasks to data-plane entities, like programmable switches or SmartNICs, has been proposed [1], [2]. Offloading allows network devices to implement packet-processing tasks that go beyond pure packet forwarding. This allows to partially or even fully offload VNFs to the network data plane.

Considering all these implementation options, network developers can choose on which kind of networking node which functionality should be implemented. A simple header modification task can, e.g., be implemented by a VM- or container-based VNF, by a P4 programmable switch, or by a SmartNIC. The additional freedom comes at the cost of increased complexity when it comes to prototyping and testing network applications — especially if complex network services composed of multiple individual network functions are considered. Furthermore, programmable hardware switches and SmartNICs are costly and might not be available to every developer, making local prototyping and testing infeasible. This motivates the need for a prototyping solution that allows to quickly implement complex networking scenarios with different node types, including offloading solutions like P4 switches and SmartNICs, and to verify that the deployment behaves as intended.

In this paper, we introduce *FOP4*, a novel prototyping platform for heterogeneous networking scenarios especially focusing on offloading functionalities. Our work substantially extends the existing Containernet platform [3], [4], which only supports container-based and VM-based VNFs and does not support offloading scenarios. More precisely, FOP4 adds support for P4 switch-based and SmartNIC-based offloading scenarios and provides a backward-compatible API to program those scenarios in the same way as Containernet scenarios are defined. Our resulting contributions, presented in this paper, are threefold: after presenting relevant related work in Section II, we describe VNF offloading scenarios on different devices and define requirements for a prototyping platform in Section III. We then introduce our novel, open-source prototyping platform, which uses network emulation concepts to be able to prototype large scenarios with networking nodes utilizing different offloading technologies. The platform runs on a single machine, e.g., a developer's laptop. We explain its design in Section IV. Finally, we evaluate the scalability of our platform and present a case study of an extension to In-band Network Telemetry (INT) [5] in Section V.

II. Related Work

Prototyping of VNFs and offloading scenarios requires execution platforms on which the virtual functions, as well as the offloaded functionality, can be deployed and tested. This can be done with full-blown testbed installations, e.g., based on DevStack [6]. The downside of testbed installations, however, is their high resource demands and slow deployment times. More lightweight, usually emulation-based prototyping platforms such as Mininet [7] or Maxinet [8], in contrast, focus more on large-scale SDN scenarios and on prototyping SDN applications rather than VNFs and offloading scenarios. Other solutions, like NIEP [9] or OSM's vim-emu [3], focus on NFV infrastructure emulation but lack support for SmartNICs or advanced SDN technologies, like P4. NIEP is even more limited by mandating the use of ClickOS-based VNFs [10]. The Kathará framework, presented by Bonofiglio et al. [11], in contrast, explicitly supports the combined deployment of P4 switches and containers while also providing a platform to quickly setup arbitrary network topologies. However, there is no support for SmartNIC prototyping and the available programming model is limited to static configuration files that make complicated building dynamic prototyping scenarios — something that can easily be done with Mininet or Containernet-based solutions [3], [4], [7].

P4, which is the reference language for network data plane programming, is used to describe and implement several use cases of VNF offloading. P4 allows the developer to describe flexible match-action processing, simple arithmetics, and a programmable parser and de-parser with a C-like language. The language is target independent: P4 can be compiled against different target devices (e.g., programmable switches [1], SmartNICs, FPGAs [12]). Furthermore, the language is also protocol independent meaning that it is unaware of any network protocols. VNF offloading scenarios using P4 are, for example, in SilkRoad [13], which uses it to implement a stateful layer-4 load balancer. In [14], an in-network caching application is presented. In [15], MapReduce aggregation is accelerated by programmable switches. Another work proposes to accelerate Neural Network inference with P4 switches and SmartNICs [16]. All of these use cases, however, implement their solutions on physical devices and testbeds. This can become costly, challenging and time consuming when bigger networks have to be tested.

While programmable switches are used to offload VNFs along the network path, SmartNICs are used to accelerate end-host functionalities. Most of the cloud computing providers are starting to adopt them in their data centers. A notable example is Microsoft [17] who massively deployed FPGA-based SmartNICs in its datacenters. SmartNICs can be programmed in multiple ways like Hardware Description Language (HDL), Micro-C [18] or even P4 [18]. Another way to exploit SmartNICs is accelerating eBPF (extended Berkeley Packet Filter) programs [19]. All the emulation platforms reported above lack support for SmartNICs abstraction, making it impossible to emulate a network with these devices.

To fill those gaps, we introduce the first prototyping platform that combines both: Container-based VNF prototyping as well as prototyping of different offloading scenarios, including P4 switches as well as SmartNICs by means of eBPF programs.

III. Prototyping Offloading Scenarios

VNF offloading has many benefits [13]–[16] but work that focuses on development support by means of simplifying the prototyping process of complex offloading scenarios is still missing. We believe that an easy-to-use, lightweight prototyping platform can be of great benefit not only for the research community but also for teaching and competence development purposes. Such a platform will lower the entry barrier and simplify the adoption of offloading technologies.

This type of prototyping platform must allow to test the functionality and correct behavior of complex use cases without creating high costs, requiring painful, time-intensive configurations of testbeds, or complicated debugging setups. It is clear, that a software-based, lightweight prototyping platform will always have tradeoffs compared to physical offloading implementations. Still, if a prototyping platform allows developers to at least verify that their implementations work as they should, without the need of costly hardware testbeds, it is already of great value. In any case, a prototyping platform will reduce turnaround and debugging times and will simplify the implementation and execution of automated functional tests. To build such a platform for offloading scenarios, we identify five major requirements:

- **Req. 1**: A prototyping platform should support different programmable network devices, such as P4-enabled switches or SmartNICs but also classic VNF technologies, like containers. The developer can, thus, test the functionality of various VNF offloading use cases in a controlled environment. Furthermore, both end-host and in-network offloading can be tested.
- **Req. 2**: A platform should allow dynamic reconfiguration of the network at runtime, making it possible to scale up or down the deployment, emulating dynamic networking scenarios typical for NFV use cases.
- **Req. 3**: A platform should reduce the configuration and deployment time and it should support easy debugging of all involved components. This reduces turnaround times and speeds up the development process.
- **Req. 4**: A platform should support real-world services and functions as they are going to be deployed in the real-world scenarios, i.e., the actual (in the best case unchanged) VNF code should be executed. This is fundamental to make the tests as similar as possible to the real physical deployment.
- **Req. 5**: Finally, a platform needs to be modular and should allow developers to extend it to add support for new devices, and new scenarios.

All the prototyping platforms considered in Section II do not fulfill one or more of these requirements. This motivated us to build FOP4, presented in the next section.

IV. FOP4: Prototyping of Heterogeneous Offloading Scenarios

FOP4 provides a rapid prototyping environment to implement VNF offloading use cases without requiring real physical systems, like P4 switches or SmartNICs. To achieve this it supports P4 switches through the reference P4 software switch developed by the P4 Language Consortium and SmartNICs through eBPF programs (**Req. 1**). The use of these software-based technologies, together with container technology to implement non-offloaded VNFs, make our platform able to emulate a realistic execution environment for VNFs. More importantly, it allows to directly move the developed and tested VNFs directly into production without large modifications (**Req. 4**).

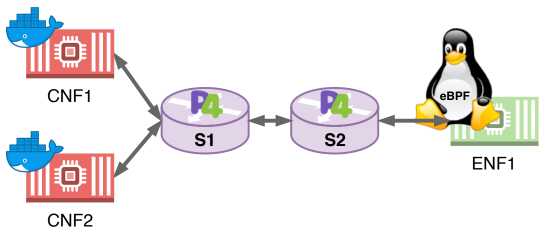


Fig. 1: FOP4 topology example with different node types defined by Listing 1

```
1  # create empty P4-Containernet network topology
2  net = FOP4()
3  # add Docker-based VNFs: CNF1 and CNF2
4  cnf1 = net.addDocker("CNF1", image="ubuntu:trusty")
5  cnf2 = net.addDocker("CNF2", image="custom:latest")
6  # add P4 switch with Thrift-based interface
7  s1 = net.addP4Switch(name="S1", netcfg=False,
8      json="example_p4.json",
9      switch_config="switch_config.cfg")
10 # add P4 switch controlled by ONOS
11 s2 = net.addP4Switch(name="S2",
12     pipeconf="org.onosproject.pipelines.basic")
13 # add emulated SmartNIC host for eBPF programs
14 enf1 = net.addEbpfHost("ENF1")
15 # add links to interconnect nodes
16 net.addLink(cnf1, s1)
17 net.addLink(cnf2, s1)
18 net.addLink(s1, s2)
19 # specify eBPF program executed in SmartNIC of ENF1
20 net.addLink(enf1, s2,
21     ebpfProgram1="./ebpf_program.o")
22 # start emulated network scenario
23 net.start()
```

Listing 1: Example FOP4 topology definition script implementing the scenario shown in Figure 1

Figure 1 shows an example of a heterogeneous offloading scenario running inside FOP4. The example contains two Docker-based VNFs (`CNF1` and `CNF2`), two P4 switches (`S1` and `S2`), and a network node running a SmartNIC emulated by an eBPF program (`ENF1`). The shown topology is defined using a simple Python script with less than 25 lines of code as shown in Listing 1. Extending the existing Mininet and Containernet topology APIs has the benefit that our proposed platform remains backwards-compatible to existing Mininet and Containernet scripts while still allowing for new node types, like P4 switches and SmartNIC hosts.

FOP4 supports a P4 switch, called BMv2 (Behavioral Model version 2), which is open source and directly maintained by the *P4 Language Consortium*. It is considered as the reference P4 software switch and it is developed to support most of the functionalities available on physical P4 switches. A program that correctly compiles and runs on BMv2 is supposed to work also on the physical equivalent (**Req. 4**). The BMv2 software switch supports also counters and registers allowing to develop stateful applications that run directly in the data plane.

To run a BMv2 switch a P4-compiled pipeline is needed. The compilation of P4 programs into such a pipeline description is done by a P4 compiler[1]. However, we do not enforce the use of a specific compiler and leave it to developers to pick the toolchain of their choice, by directly using the already compiled pipelines in our platform. On top of the compiled pipeline, then, the switch table configuration has to be deployed. This is the configuration of the switch that can also be modified at runtime. The BMv2 P4 software switch can be controlled and programmed using two different interfaces: P4Runtime[2], and a Thrift-based interface. P4Runtime allows to control a P4 switch using ONOS. The Thrift-based interface, instead, allows to program and modify the configuration of the switch with a Command Line Interface (CLI). Our platform supports both of these interfaces. The two interfaces can coexists, even if the P4Runtime interface is selected to configure the switch, the CLI Thrift-based interface is still available and it can be used to interact with it. Depending on the selected interface, there are two different ways to configure the switch table and to deploy the P4-compiled pipeline. If the P4Runtime interface is selected, and thus ONOS is elected as controller, the developer has to define the configuration of the pipeline (*Pipeconf* in ONOS terminology). When the switch is booting, it contacts the *ONOS Network Configuration Service*, which provides the P4-compiled pipeline and subsequently the table switch configuration. From then on, the switch is controlled either by ONOS or by the CLI. If, instead, the Thrift-based CLI interface is selected, the developer has to specify in advance the P4-compiled pipeline and the switch table configuration. These two parameters can be specified either as one of the parameters of the switch when configuring it, or configuring it manually after the switch is booted. An example of the two configuration modes is available in Listing 1 (lines 6 to 12), showing how P4 switches can be deployed in an emulated network using our platform.

To emulate SmartNICs, we exploit eBPF, which allows to develop restricted C code that can run directly in the Linux kernel offering similar functionalities as SmartNICs do. With eBPF, a developer can implement packet-filtering applications or even more complex packet processing tasks that directly run in kernel space. Selecting eBPF programs to emulate SmartNICs is based on two reasons. First of all, SmartNICs are already used to accelerate eBPF programs, allowing to offload programs directly on the programmable NIC [19]. Secondly, a P4-to-eBPF compiler [20] is available; in this way developers do not need to learn different languages and technologies to deploy offloading use cases. If they just know the P4 language, they can offload computation to either SmartNICs or switches. To run an emulated SmartNIC, a compiled eBPF program is needed. The compilation can be done with or without the P4-to-eBPF compiler. Again, we leave the choice of the used technology to the developer and support the direct use of already compiled eBPF programs. The only limitation that we introduce is that the eBPF program has to be an eXpress Data Path (XDP); we selected this type of program because it is

[1]P4 BMv2 Compiler https://github.com/p4lang/p4c
[2]P4Runtime repository https://github.com/p4lang/PI

supported by the P4 to eBPF compiler. XDP is a specific class of eBPF programs that is deployed in the lowest level of the networking stack, providing maximum performance in packet processing. To support this kind of device in our platform, we extended the standard *Mininet* host, injecting an eBPF program on one of the interfaces that are running on the host itself. The only parameter needed to run an eBPF program inside our platform is the path of the program. An example is available in Listing 1 (line 14). First of all, the developer has to define a special kind of host called *eBPFHost*. When a link is created from this host, then the developer can specify which eBPF program has to be injected on the interface connected to that link (Listing 1 lines 20 and 21). Therefore, we even support adding multiple eBPF programs to a single host in the emulated network topology.

To fulfill **Req. 2**, FOP4 also supports adding new interfaces at runtime to the P4 switch through the Python APIs. It also supports adding links to eBPF hosts, allowing to deploy SmartNICs when the emulated network is already running. Finally, being based on *Mininet*, our platform is easy to use, debug and extend, supporting **Req. 3** and **Req 5**. To further simplify VNF offloading use cases, we also extend the *MiniEdit* GUI, adding support for the newly introduced network devices. In this way, developers can deploy their own emulation environment, accessing the switch CLI from the GUI rather than using the platform's Python APIs.

FOP4 is available to the community as open source project[3].

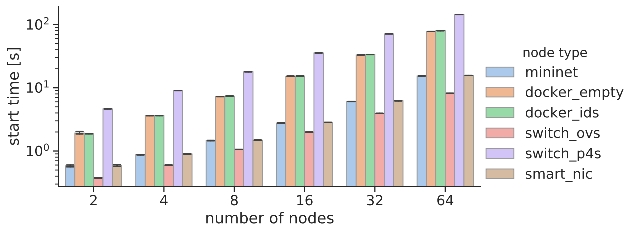
V. Evaluation

We split the evaluation of our novel prototyping platform into two parts. First, we evaluate the platform as such in terms of startup times, i.e., the time it takes to bootstrap and run scenario prototypes, as well as in terms of scalability, i.e., how much of resources the platform needs, in Section V-A. Second, we performed a case study with an INT [5] scenario in Section V-B. This case study demonstrates how the presented platform can be used to prototype complex networking scenarios containing P4-based, SmartNIC-based, and container-based VNFs.

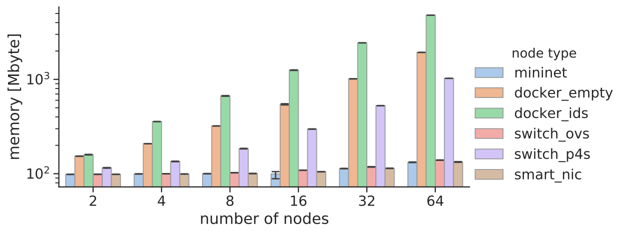
A. Platform Performance and Scalability

To evaluate the performance and scalability of our platform, we designed an experiment in which we emulate network scenarios of different sizes and with different node types. More specifically, a scenario of size $S \in [2, 4, 8, 16, 32, 64]$ is a homogeneous deployment of S nodes, where each node can be either a default Mininet host [7], an empty Docker container [4], a Docker container running a Suricata IDS [21], an OVS switch [22], a BMv2 P4 switch, or a host that emulates SmartNICs by executing an eBPF program. In each scenario, each of the S nodes is connected to a single OVS switch, resulting in a star-like topology. Heterogeneous deployments, in which multiple node types are mixed, are also possible. We use homogeneous scenarios to better compare the impact

[3]FOP4 repository: https://github.com/ANTLab-polimi/FOP4



(a) Platform start times



(b) Platform memory consumption

Fig. 2: Platform performance results for different node types and scenarios with up to 64 nodes

of the node type on overall performance. All experiments are executed on a machine with Intel(R) Xeon(R) W-2145 CPU at 3.70 GHz and 32 GB of memory. Each experiment was repeated 50 times and the error bars indicate 95 % confidence intervals.

The first experiment evaluates the time needed to start (bootstrap, configure, and run) the platform with the given scenario. Figure 2a shows the results and the linear relationship between the start times and the size of the scenarios. It also shows that starting OVS or Mininet hosts is much faster than starting Docker-based nodes or P4 switches. A user should thus only use complex nodes, like Docker containers or P4 switches, if needed to implement the prototyped functionality. Still, even in the worst case, with 64 P4 switches in a scenario, the platform can be started in less than 146 s, allowing quick turnaround times even when complex P4 scenarios are prototyped.

In the second experiment, we focus on the memory footprint of our platform. We use the same experimental setup and executed the different scenarios with up to 64 nodes. Figure 2b proofs that the memory usage of our platform also has a linear relationship with scenario size. It shows that Mininet, OVS, and SmartNIC nodes need far less memory than the other nodes. The Docker nodes that run the IDS system use the most memory, which is expected and heavily depends on the used VNF application running in the container. Still, the largest scenario with 64 instances of the IDS VNF does not need more than 4.84 Gbyte of memory and can thus be executed on every common-off-the-shelf laptop. Both results highlight the main benefits of the proposed platform, allowing to quickly prototype complex networking scenarios that comprise containers VNFs, P4 switches, and SmartNIC solutions on a developer's laptop.

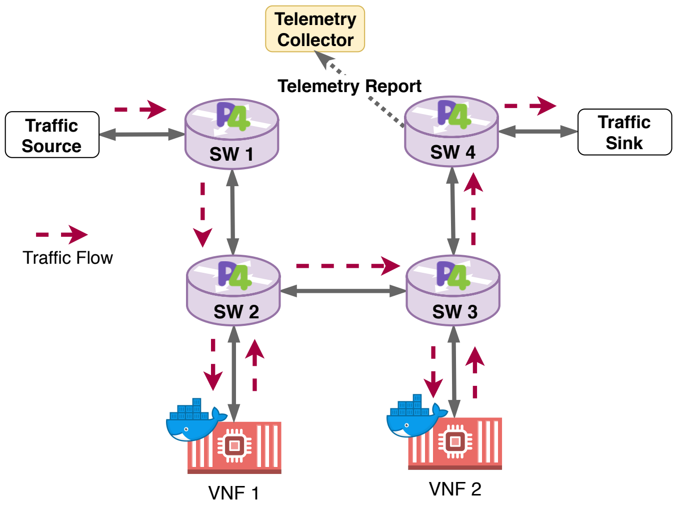


Fig. 3: Topology for the case study on eINT

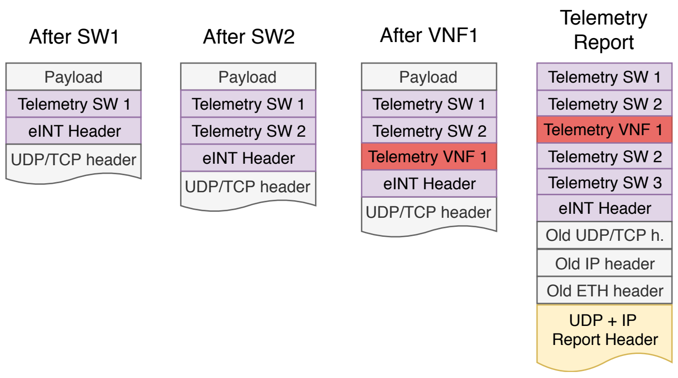


Fig. 4: Examples of how the packets are modified by switches and VNFs, and example of a telemetry report

B. *Case Study: extended In-band Network Telemetry*

Finally, we present the extended In-band Network Telemetry (eINT) use case. It demonstrates (1) how our platform can be used to prototype complex networking scenarios containing different devices and (2) mixing docker-based VNFs and P4 devices.

In-band Network Telemetry [5] is a framework that allows to collect and report the network state without continuously polling the network devices and without the direct intervention of the control plane. The key point is the injection of a new INT header directly into the packets flowing in the data plane. Network telemetry information can be generated and collected without creating any new packets. The INT header contains the *telemetry instruction* that is interpreted by the INT-enabled network devices. Those devices collect the required telemetry information and write it into the packet itself. An INT-enabled network contains three different devices:

- **INT-source**: it embeds the *telemetry instruction* and initializes the INT header (`SW1` in Figure 3);
- **INT-transit**: it collects and pushes the telemetry information inside the packet (`SW1`, `SW2`, `SW3`, `SW4`, `VNF1`, `VNF2`, in Figure 3, are all INT-transit nodes);
- **INT-sink**: it retrieves the collected results from the packet, removes the INT header and generates the *telemetry report* sending it to the telemetry collector (`SW4` in Figure 3).

The full standard is defined by the *P4 Language Consortium* and it is available at [23].

The case study we present is an extension to this framework. eINT includes not only telemetry information from network data plane devices (e.g., P4 switches), but also from Network Functions (e.g., Docker containers). In detail, we extend the instruction part of the INT header, dedicating the eight least significant bits of the 16-bit instruction bitmap to VNF telemetry instructions. Those bits are unused in the current standard (apart from the last bit which is used as a checksum complement). The VNF-INT instruction bits are then encoded as a bitmap:

- bit 8 (MSb): Virtual Network Function ID;
- bit 9: CPU usage;
- bit 10: memory usage;
- bit 11: timestamp in VNF;
- bit 12-14: unused (they can be used for other instructions);
- bit 15: Checksum Complement (part of the INT standard).

The other extension we make to the standard is the amount of bytes each INT-transit node is adding. While with the INT standard, the amount of bytes added by each nodes is related to the amount of bits set in the INT instruction field, with our extension we also need to consider the bits of the VNF-INT instruction bits. Each transit node has to add an amount of bytes that is the logical OR between these two instruction bitmaps, considering that each bit means adding four or eight bytes to the packet header.

The collected statistics can also be extended or modified by updating the configuration of the INT-source node. With eINT, thus, a developer can track the behavior of a VNF packet by packet. This can be useful, for example, to scale up or down a Network Function when the CPU or memory usage is above a certain threshold. The telemetry information can also be helpful to better understand the behavior of a VNF chain by tracking the time taken by each VNF to process a packet or by checking if a specific packet is actually traversing the right VNF chain and no packet "leaks" are happening.

Figure 3 shows an example of a topology in which both data plane P4-enabled devices and Docker VNFs with support for INT are deployed. With FOP4, the deployment of a service chain with eINT support is made easy. The platform supports P4-enabled switches as well as VNFs as Docker containers. Once the P4 program and the Docker container is configured, developers only have to think about which statistics they want to collect, then all the network deployment is a single Python script that exploits the API described in Section IV.

Figure 4, instead, shows how the packet is modified by the nodes in the network of the topology in Figure 3. In particular, we can see that `SW1`, which is an INT-source node, is adding the eINT header and the telemetry of the node itself (remember that in our case `SW1` works also as INT-transit node). We can also see that `VNF1` behaves like `SW2`. Since they are INT-transit nodes, they are both adding their telemetry information. Finally, a telemetry report is shown. This report is sent by the INT-sink (`SW4` in Figure 3) to the telemetry collector for further elaboration and statistics extraction. The report, which is also standardized [24], contains all the information about the packet (without the L4 payload) plus all the telemetry added by the INT-transit nodes.

To demonstrate the feasibility of eINT in FOP4, we provide a proof-of-concept (POC) with the topology shown in Figure 3. In this POC, we limit INT-enabled VNF to be only INT-transit nodes. Each INT-enabled VNF is in charge of interpreting the VNF-INT instruction bitmap and adding the requested statistics to the packet. In our POC the program running in each VNFs is implemented as a Python script exploiting the Scapy library, but the implementation can be also accelerated using eBPF programs. This eINT POC is available on GitHub[4].

VI. Conclusion

We presented FOP4, a novel prototyping platform to create and test NFV offloading scenarios on heterogenous nodes, like P4-switches and SmartNICs. This is not possible with any existing NFV prototyping solution. The easy-to-use Python interface of our platform allows developers to quickly setup complex network scenarios, such as the presented eINT use case. These scenarios can be deployed within seconds on limited resources, as shown in our evaluation. We argue that the quick turnaround times and removed need for costly testbeds has the potential to substantially simplify the adoption and use of offloading technologies by network function developers.

The presented platform as well as the eINT use case are made available to the community as open source projects.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No. H2020-ICT-2016-2 761493 (5GTANGO), and the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

References

[1] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 99–110, 2013.

[2] S. Pontarelli, R. Bifulco, M. Bonola, C. Cascone, M. Spaziani, V. Bruschi, D. Sanvito, G. Siracusano, A. Capone, M. Honda *et al.*, "Flowblaze: Stateful packet processing in hardware," in *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19). USENIX Association*, 2019.

[3] M. Peuster, H. Karl, and S. van Rossem, "MeDICINE: Rapid Prototyping of Production-ready Network Services in Multi-PoP Environments," in *2016 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2016, pp. 148–153.

[4] M. Peuster, J. Kampmeyer, and H. Karl, "Containernet 2.0: A rapid prototyping platform for hybrid service function chains," in *2018 4th IEEE Conference on Network Softwarization and Workshops (NetSoft)*, June 2018, pp. 335–337.

[5] C. Kim, A. Sivaraman, N. Katta, A. Bas, A. Dixit, and L. J. Wobker, "In-band network telemetry via programmable dataplanes," in *ACM SIGCOMM*, 2015.

[6] OpenStack Project, "DevStack," online at: https://docs.openstack.org/devstack/latest/, 2018.

[7] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: rapid prototyping for software-defined networks," in *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*. ACM, 2010, p. 19.

[8] P. Wette, M. Draxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "Maxinet: Distributed emulation of software-defined networks," in *Networking Conference, 2014 IFIP*. IEEE, 2014, pp. 1–9.

[9] T. N. Tavares, L. da Cruz Marcuzzo, V. F. Garcia, G. V. de Souza, M. F. Franco, L. Bondan, F. D. Turck, L. Z. Granville, E. P. D. Junior, C. R. P. dos Santos, and A. E. Schaeffer-Filho, "Niep: Nfv infrastructure emulation platform," in *2018 IEEE 32nd International Conference on Advanced Information Networking and Applications (AINA)*, May 2018, pp. 173–180.

[10] J. Martins, M. Ahmed, C. Raiciu, V. Olteanu, M. Honda, R. Bifulco, and F. Huici, "Clickos and the art of network function virtualization," in *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 459–473. [Online]. Available: http://dl.acm.org/citation.cfm?id=2616448.2616491

[11] G. Bonofiglio, V. Iovinella, G. Lospoto, and G. Di Battista, "Kathar: A container-based framework for implementing network function virtualization and software defined networks," in *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, April 2018, pp. 1–9.

[12] S. Ibanez, G. Brebner, N. McKeown, and N. Zilberman, "The p4-¿ netfpga workflow for line-rate packet processing," in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, 2019, pp. 1–9.

[13] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 2017, pp. 15–28.

[14] X. Jin, X. Li, H. Zhang, R. Soulé, J. Lee, N. Foster, C. Kim, and I. Stoica, "Netcache: Balancing key-value stores with fast in-network caching," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 121–136.

[15] A. Sapio, I. Abdelaziz, A. Aldilaijan, M. Canini, and P. Kalnis, "In-network computation is a dumb idea whose time has come," in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 150–156.

[16] D. Sanvito, G. Siracusano, and R. Bifulco, "Can the network be the ai accelerator?" in *Proceedings of the 2018 Morning Workshop on In-Network Computing*. ACM, 2018, pp. 20–25.

[17] D. Firestone, A. Putnam, S. Mundkur, D. Chiou, A. Dabagh, M. Andrewartha, H. Angepat, V. Bhanu, A. Caulfield, E. Chung *et al.*, "Azure accelerated networking: Smartnics in the public cloud," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018, pp. 51–66.

[18] Netronome, "Programming NFP with P4 and C," online at: https://www.netronome.com/m/documents/WP_Programming_with_P4_and_C.pdf, 2018.

[19] J. Kicinski and N. Viljoen, "ebpf hardware offload to smartnics: cls bpf and xdp," *Proceedings of netdev*, vol. 1, 2016.

[20] W. Tu, F. Ruffy, and M. Budiu, "P4C-XDP: Programming the Linux Kernel Forwarding Plane using P4," in *Linux Plumbers Conference*, 2018.

[21] OISFoundation. (2019) Suricata: Open Source IDS / IPS / NSM engine. [Online]. Available: https://suricata-ids.org/

[22] The Linux Foundation. (2019) Open vSwitch. [Online]. Available: https://www.openvswitch.org/

[23] P4 Language Consortium — Applications Working Group, "In-band Network Telemetry (INT) Dataplane Specification," online at: https://github.com/p4lang/p4-applications/blob/master/docs/INT.pdf.

[24] P4 Language Consortium — Applications Working Group, "Telemetry Report Format Specification," online at: https://github.com/p4lang/p4-applications/blob/master/docs/telemetry_report.pdf.

[4]https://github.com/ANTLab-polimi/FOP4/tree/master/P4_examples/eINT