A Framework for eBPF-Based Network Functions in an Era of Microservices

Sebastiano Miano, *Member, IEEE*, Fulvio Risso, *Member, IEEE*, Mauricio Vásquez Bernal, Matteo Bertrone, and Yunsong Lu

***Abstract*—By moving network functionality from dedicated hardware to software running on end-hosts, Network Functions Virtualization (NFV) pledges the benefits of cloud computing to packet processing. While most of the NFV frameworks today rely on kernel-bypass approaches, no attention has been given to kernel packet processing, which has always proved hard to evolve and to program. In this article, we present Polycube, a software framework whose main goal is to bring the power of NFV to in-kernel packet processing applications, enabling a level of flexibility and customization that was unthinkable before. Polycube enables the creation of arbitrary and complex network function chains, where each function can include an efficient in-kernel data plane and a flexible user-space control plane with strong characteristics of isolation, persistence, and composability. Polycube network functions, called Cubes, can be dynamically generated and injected into the kernel networking stack, without requiring custom kernels or specific kernel modules, simplifying the debugging and introspection, which are two fundamental properties in recent cloud environments. We validate the framework by showing significant improvements over existing applications, and we prove the generality of the Polycube programming model through the implementation of complex use cases such as a network provider for Kubernetes.**

***Index Terms*—NFV, eBPF, XDP, linux.**

I. Introduction

With the advent of Software Defined Networks (SDN) and Network Functions Virtualization (NFV), a large number of Network Functions (NFs)[1] are becoming pure software images executed on general-purpose servers, running either as virtual machines (VMs) or as cloud native software. Possible examples include load balancing [1], [2], [3], [4], congestion control [5], [6], [7], and application-specific network workloads such as DDoS Mitigation [8] or key-value stores [9], [10]; thanks also to the development and availability of programmable network devices (e.g., SmartNICs) [11]. The increased flexibility (software is intrinsically easier to program compared to the hardware), and the recent advances in terms of speed for the software packet processing have then contributed to the proliferation of a myriad of VNFs frameworks that provide implementations of efficient and easily programmable software middleboxes [12], [13], [14], [15], [16], [17], [18].

Current solutions to implement the dataplane of those *software packet processing* applications rely mostly on kernel bypass approaches, by giving to the user-space direct access to the underlying hardware (e.g., DPDK [19], netmap [20], FD.io [21]) or by following a *unikernel* approach, where only the minimal set of OS functionalities, required for the application to run, are built with the application itself (e.g., ClickOS [22], [23], [24]). These approaches have perfectly served their purposes, with efficient implementations of software network functions that have shown potential for processing 10-100Gbps on a single server [25], [26], [27].

Recently, new technologies such as 5G, edge computing and IoT among the others, led to a significant increase in the total number of connected devices and consequent network load, hence originating two new trends. From one side, traditional "centralized" appliances (e.g., global datacenter firewalls) are hard to scale, leading to a more distributed approach in which network functions are implemented directly on end hosts (e.g., datacenter servers). From the other side, cloud-native technologies are used to built NFs packaged in containers, deployed as microservices, and managed on elastic infrastructure through agile DevOps processes and continuous delivery workflows [28]. These new requirements have also caused a visible change in the type and requirements of network functionalities deployed across the data center. Network applications should be able to continuously adapt to the runtime behavior of cloud-native applications, which might regularly change or be scheduled by an orchestrator, or easily interact with existing "native" applications by leveraging kernel functionalities - all of this without sacrificing performance or flexibility. For instance, cloud-native platforms, like Kubernetes [29], can exploit different network plug-ins[2] to implement the underlying data plane functionalities and transparently steer packets between micro-services.

Manuscript received June 1, 2020; revised October 28, 2020 and January 4, 2021; accepted January 25, 2021. Date of publication January 29, 2021; date of current version March 11, 2021. The associate editor coordinating the review of this article and approving it for publication was T. Zinner. *(Corresponding author: Sebastiano Miano.)*

Sebastiano Miano, Fulvio Risso, Mauricio Vásquez Bernal, and Matteo Bertrone are with the Department of Control and Computer Engineering, Politecnico di Torino, 10129 Torino, Italy (e-mail: sebastiano.miano@polito.it; fulvio.risso@polito.it; mauricio.vasquez@polito.it; matteo.bertrone@polito.it).

Yunsong Lu was with the Networking and Emerging Technologies Group, Futurewei Technologies, Inc., Santa Clara, CA 95050 USA. He is now with the Linux Foundation, San Francisco, CA, USA (e-mail: yunsong.lu@futurewei.com).

Digital Object Identifier 10.1109/TNSM.2021.3055676

[1]This article uses the term Network Function (NF) to specify components such as traditional individual appliances (e.g., bridge, router, NAT, etc.), while a *service* defines a more complex scenario in which multiple NFs must cooperate, e.g., through a *chain* of NFs.

[2]A list of network plugins (also known as Container Network Interface (CNI) plug-ins) is available in the Kubernetes Cluster Networking page, https://kubernetes.io/docs/concepts/cluster-administration/networking/.

Unfortunately, the previously mentioned kernel-bypass approaches suffer within this new scenario [30]. First, they require the exclusive allocation of resources (i.e., CPU cores) to achieve good performance; this is perfectly fine when we have machines dedicated to networking purposes but it becomes overwhelming when this cost has to be paid for every server in the cluster since they permanently steal precious CPU cycles to other application tasks. Second, they require to re-implement the entire network stack in userspace, losing all the well-tested configuration, deployment and management tools developed over the years within the operating system. Third, they rely on custom or modified versions of NIC (network interface card) drivers, which may not be available on public cloud platforms, also requiring a non-negligible maintenance cost. Last but not least, they have difficulties (and poor performance) when dealing with existing kernel implementations or communicating with applications that are not implemented using the same approach, requiring them to adhere to custom-defined APIs (e.g., mTCP [31]) or to change the original application logic (e.g., StackMap [32]). As a consequence, most of the existing cloud-native network plug-ins for Kubernetes still rely on functionalities and tools embedded into the operating system network stack (e.g., iptables, ipvs, linuxbridge), while Polycube (and recently, Cilium) being notable exceptions. Unfortunately, the drawbacks of this approach are also evident. First of all, *fixed* kernel network applications are notoriously slow and inefficient given their generality, which impairs the possibility to specialize them depending on workloads or the type of application that is running on top. Secondly, software network functions (or the associated kernel modules [33]) that live in the kernel have also proven hard to evolve due the complexity of the code and the difficulties in maintaining, up-streaming or modifying it.

This article proposes **Polycube**, a novel software framework that addresses these deficiencies by adopting a different approach for building and running NFs. Polycube exploits the extended Berkeley Packet Filter (eBPF), whose characteristics are presented in Section II, to build the data plane of the NFs and to provide the flexibility and easy development process that were almost impossible to achieve in "legacy" kernel implementations. Polycube enables the creation of efficient, modular and dynamically re-configurable network functions that are executed within the Linux kernel; the above components can be easily chained to create more complex services.

Overall, this article makes the following contributions:

- We show the design and architecture of Polycube (Section IV), which moves to the kernel space some ideas and concepts already established in other NFV frameworks, hence with different challenges and solutions.
- We present the mechanism employed by Polycube to enable the creation of arbitrary chains of in-kernel network functions (Section V), which mimics the well-known model in use in the physical world that deploys *functions* (e.g., bridges, routers) connected by *wires*.
- We provide a description of the Polycube programming model, including the APIs and abstractions used to simplify the development of new NFs (Section VI).
- We introduce a generic model for the control and management plane of each NF that is used to simplify the manageability and accelerate the development of new network services (Section VII), which complement the *dataplane-only* approach proposed by eBPF.
- Finally, we identify and quantify both the overhead introduced by the Polycube programming model and its the data plane abstractions and the performance improvements brought to existing kernel implementations (Section IX).

Polycube is open-source and available at [34].

II. Background

This section lists the main properties that we believe are fundamental in today's environments (Section II-A), it provides a brief overview of the extended Berkeley Packet Filter subsystem (Section II-B) and why we believe this may represent a good choice for networking applications (Section II-C).

A. Desired Properties for Cloud-Native NFs

Low Overhead: Although efficiency is always a desirable property, this assumes even more significance in the cloud-native context where servers are mostly used to deliver high-level services (e.g., Web portals, databases, etc.) and networking components are often perceived as an unavoidable overhead, particularly within edge clouds, where the number of available resources is limited.

Agile Development: Newer software data planes should follow the same *continuous delivery* (CD) software development typical of microservices, making it possible to easily update the existing application by providing a replacement that does not disrupt the typical service workflow [35].

Runtime Flexibility and Optimizations: From a developer's point of view, writing an efficient and, at the same time, easy to maintain software data plane is a daunting task. Most of the time is just a matter of finding the right trade-off between *simplicity* (e.g., modularity, easy-to-read code) and *performance*. We often see application-specific, and ad-hoc techniques applied only to particular use-cases [26], [27], which do not perform well on other scenarios or for a broader spectrum of applications. To better adapt to this new extremely dynamic environment, network components should be able to automatically adapt themself to the runtime condition with the minimum amount of programming effort.

Co-Existence With "Traditional" Ecosystem: Applications are now running on the host operating system that is shared between the different components (e.g., containers), which in turn rely on existing kernel functionality to accomplish their tasks [36]. It is then crucial that network functions can easily interact with existing "native" applications and also leverage kernel functionalities (e.g., TSO, skb metadata, etc.), without sacrificing performance and flexibility [37].

B. Extended Berkeley Packet Filter (eBPF)

The Berkeley Packet Filter (BPF) is an in-kernel virtual machine for packet filtering that has been deeply revisited starting from 2013 and is now known as extended BPF (eBPF). In addition to several architectural improvements, eBPF introduces the capability of handling generic event processing in the kernel, JIT compiling for increased performance, stateful processing using maps, and libraries (helpers) to handle more complex tasks, available within the kernel.

eBPF programs can be either written using eBPF assembly instructions and converted to bytecode using `bpf_asm` utility or in restricted C and compiled using the LLVM Clang compiler. The bytecode can then be loaded using the `bpf()` system call. A loaded eBPF program follows an event-driven architecture and it is therefore hooked to a particular type of event. Each occurrence of the event will trigger its execution, and, based on the type of event, the program might be able to alter the event context. For networking purposes, program execution is triggered by the arrival of a packet. Two hooks are available to intercept packets and possibly mangle, forward or drop them: eXpress Data Path (XDP) [38] and Traffic Control (TC). XDP programs intercept RX packets right out of the NIC driver, possibly before the allocation of the Linux socket buffer (`sk_buff`), allowing, e.g., early packet drop. TC programs intercept data when it reaches the kernel traffic control function, either in RX or TX direction. We refer the reader to [39], [40] for a more in-depth understanding of the eBPF subsystem.

C. Choosing eBPF for Network Functions

While eBPF is not the only technology proposed for high-speed and flexible packet processing (e.g., to limit our view to the most alternatives with strong industry support, DPDK [19] and FD.io [21]), the authors believe this technology is a valid alternative for the following main reasons.

Easy Development Process: eBPF (data plane) programs follow the same development process of userspace applications and can be created independently from the kernel development process, without explicit use of kernel-level primitives.

Dynamic Update: eBPF programs are dynamically injected into the kernel using the `bpf()` system call, without having to install custom kernel modules or relying on modified kernels. Dynamic (hence, at run-time) updates enable the creation of data plane software that is tailored to the actual requirements in terms of applications and workloads [41], as opposed to static kernel implementations.

High Throughput: Although being dynamically injected in the kernel, eBPF programs are executed natively on the target platform,[3] which provides a notable speed-up compared to an interpreted execution. Moreover, the execution of these programs at the XDP level enables a high-speed packet processing and forwarding, with performance also close to user-level approaches [38].

[3]By default, eBPF programs are JIT-compiled into native machine code before being executed, as the eBPF JIT flag is active by default in latest kernel releases.

Excellent Performance/Efficiency Trade-Off: eBPF programs are triggered only when a packet is received, hence do not consume CPU cycles when there is not traffic waiting to be processed, as opposed to polling-based approaches (e.g., DPDK), where the CPU usage is always 100% even in absence of traffic.

Integration With the Linux Subsystems: eBPF programs cooperate with the kernel TCP/IP stack, interact with other kernel-level data structures (e.g., FIB or neighbor table), and leverage kernel functionalities (e.g., TSO, skb metadata, etc.), possibly complementing existing networking features. Legacy applications or debugging tools can continue to be used without any change to the existing applications.

Security: As opposed to custom kernel modules, eBPF programs cannot harm the system. The in-kernel *verifier* ensures that all the operations performed inside the eBPF programs are correct and safe, discarding the injection of faulty programs.

Explicit Separation Between Stateless and Stateful Primitives: eBPF programs have a clear distinction between stateless and stateful portions of the code. Operations outside the eBPF environment are carried out only through specific "helper" functions that have a well-defined syntax and behavior. Several recent analysis frameworks [42], [43], [44] require this separation to simplify the analysis of NF operations to find buggy development semantic behavior (verifier safety does not imply "semantic safety") or to analyze the performance of a NF [45], [46]. eBPF programs have this separation as part of the original design, facilitating the adoption of the above type of analysis and concepts.

III. Design Goals and Challenges

The main objective of Polycube is to provide a common framework to network function developers to bring the power and innovation promised by NFV to the world of in-kernel packet processing, which is possible thanks to the introduction of eBPF. However, eBPF was not created with this goal in mind; it serves only as a generic virtual machine that enables the execution of user-defined program into the kernel, attaching them to specific points into the Linux TCP/IP stack or to generic kernel functions (e.g., kprobes). Then, Polycube aims to chase the following objectives, which are not covered by the eBPF subsystem.

G1 (Common Structure and Abstractions of in-Kernel NFs): The realization of *complex* network functionalities are not always possible in eBPF, given its security model that is forced by the in-kernel verifier to ensure that the execution of the program does not harm the system [47]. For instance, no abstractions currently exist to enable the concept of virtual ports from which the traffic is received or sent out, or to implement the (complex) control plane of a NF, forcing developers to dedicate a considerable amount of time to handle common control plane operations (e.g., user-kernel interaction). Polycube must provide a common programming framework and models to allow developers to use high-level abstractions to solve common problems or known limitations of eBPF in a efficient and transparent way, while the framework optimizes the implementations of those abstractions.

G2 (Programmable and Extensible NF Chaining): In a NFV environment, a packet is typically processed by a sequence of NFs, giving the possibility to run different NFs at the same time, which are also manufactured by multiple vendors. Polycube must enable the possibility to create chain of NFs in the kernel, guaranteeing the correct forwarding sequence and same degree of isolation between them.

G3 (Simple Management and Execution of the NFs): Polycube must allow external operators (e.g., SDN controllers, orchestrators, network administrators) to configure in-kernel functionalities to support a diverse set of use cases. This implies the possibility to compose and configure the datapath functionalities or to dynamically upgrade or substitute a given NF at runtime. A clear separation between the in-kernel data plane and the control plane would be desired, so that is becomes easier to dynamically regenerate and reconfigure the data path to implement the user policies.

G4 (Simple Development of Control and Management Plane): The existence of an API that allows to control the VNF behavior and its networking intent is fundamental and requires a non-negligible development effort. This task is often underestimated by most of the NFV frameworks, which mainly focus on the data plane design and performance, leaving to custom and often non interoperable control plane implementations of the NF APIs. Polycube should provide a common structure of the control plane with a *program-dependent* API that is automatically generated for each NF, freeing the developer of this additional implementation burden.

IV. Architecture Overview

This section introduces the main ideas that inspired the design of Polycube, its overall software architecture, and the most significant implementation details. An high-level description of the Polycube architecture is shown in Figure 1.

A. Unified Point of Control

All network functions within Polycube feature a unified point of control, which enables the configuration of high-level directives such as the desired service topology. In addition, it facilitates the provisioning of cross-network function optimizations that could not be applied with separately managed NFs. Polycube supports this model through a single, service-agnostic, userspace daemon, called `polycubed`, which is in charge of interacting with the different network function instances. Each different type of virtual network function in Polycube is called `Cube`.[4] A Cube can be plugged into the framework *at run-time*, upon a registration phase in which the new cube shares with `polycubed` the information required for its identification, such as its name, description or the minimum kernel version required to run the NF.

When the NF is registered, different instances of it can be created by contacting `polycubed`, which acts mainly as a proxy; it receives a request from a northbound `REST` interface and forwards it to the proper NF instance, returning back the answer to the user.

[4]In the rest of this article, we will use the term NF and Cube interchangeably. In Polycube a NF is indeed represented as a single Cube.

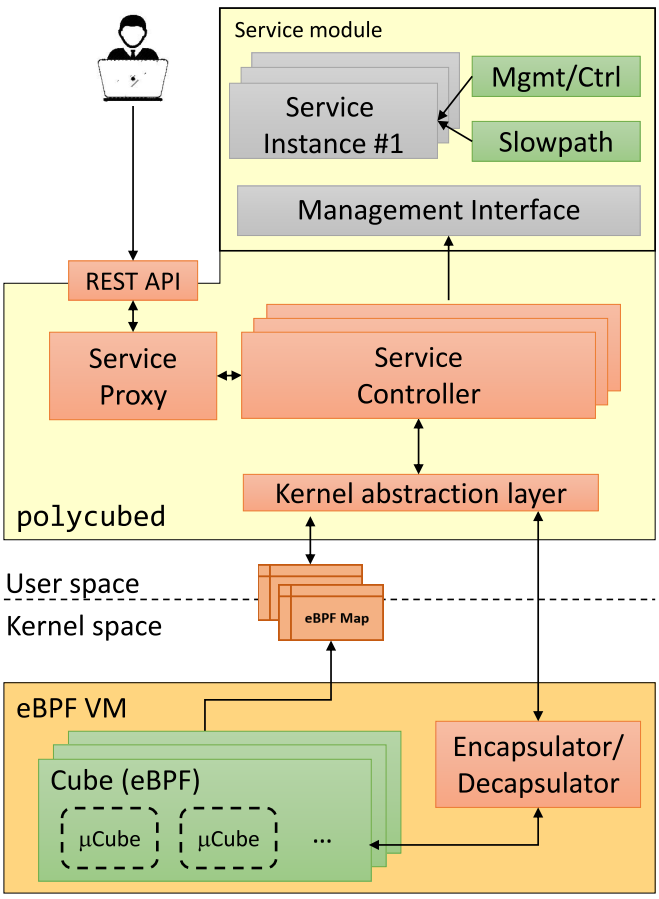


Fig. 1. The Polycube architecture.

B. Overall Structure of Polycube NFs

Each Cube is made up of a *control plane* and a *data plane*. The *data plane* is responsible for per-packet processing and forwarding, while the *control and management plane* is in charge of NF configuration and non-dataplane tasks (e.g., routing protocols).

1) Data Plane: The data plane portion of a NF is executed *per packet*, with the consequent necessity to keep its cost as small as possible. The data plane of a Polycube NF is composed by a *fast path*, namely the eBPF code that is injected into the kernel, and a *slow path*, which handles packets that cannot be fully processed in the kernel or that would require additional operations, slowing down the processing of the other packets.

Fast Path: When triggered, the fast path retrieves the packet and its associated meta-data from the receive queues, then it executes the injected eBPF instructions. Typical operations are usually very fast, such as packet parsing, lookups in memory (e.g., to classify the packet), and map updates, such as storing data in memory (e.g., statistics), for further processing. When those operations are carried out, the fast path returns a forwarding decision for that particular packet or send it to the slow path for further processing.

A Cube's fast path is composed of a single or a set of micro-blocks, called *micro-cubes (µCubes)*, which represent the smallest programming unit of the Cube's data plane. A

μCube is a single eBPF program that lies under the umbrella of single Cube, which provides a unique control plane and slow path module that is shared among all its μCubes. To form the Cube's fast path, several μCubes can be stitched together and injected separately from the Cube's control plane using the Polycube service-independent APIs.

This modular design is particularly useful because it allows developers to handle each feature separately, enabling the creation of loosely coupled NFs with different functionalities (e.g., packet parsing, classification, field modification) to be dynamically composed and replaced; each single μCube can be substituted at runtime with a different version or can be directly removed from the chain if its features are not needed anymore. Second, it can be useful to overcome some well-know eBPF limitations such as the maximum size of an eBPF program or the inability to create unbounded loops in the code. Furthermore, chains of μCubes do not introduce any noticeable overhead, as presented in Section IX-C2.

This design choice introduces the necessity to specify an order of execution of the μCubes inside the NF; when a packet reaches a Cube composed of different micro-blocks, Polycube has to know the first module to execute, which in turn will trigger the execution of the others μCubes within an arbitrary order based on its internal logic. To do this, Polycube introduces the concept of `HEAD` μCube, which is unique within the Cube itself and represents the entry point of the entire NF, whose execution is triggered upon the reception of a packet in a port. Then, it can "jump" to the others μCubes.

Slow Path: Although eBPF offers the possibility to perform some complex and arbitrary actions on packets, it suffers from some well-known limitations due to its restricted environment, which however are necessary to guarantee the integrity of the system. Those limitations may impair the flexibility of the network function, which *(i)* may not be able to perform complex actions directly in the eBPF fast path or *(ii)* could slow down its execution, adding more instructions in the fast path to handle exceptional cases. To overcome those limitations, Polycube introduces an additional data plane component that is no longer limited by the eBPF virtual machine and it can hence execute arbitrary code. The *slow path* module is executed in userspace and interacts with the eBPF fast path using a set of components provided by the framework. The eBPF fast path program can redirect packets (with custom meta-data) to the slow path, similar to Packet-In messages in OpenFlow. Similarly, the slow path can send packets back to the fast path; in this case, Polycube provides the possibility to inject the packet into the *ingress* queue of the network function port, simulating the reception of a new packet from the network, or into the *egress* queue, hence pushing the packet out of the network function.

2) Control and Management Plane: The control plane of a virtual network function is the place where out-of-band tasks, needed to control the data plane and to react to possible complex events (e.g., Routing Protocols, Spanning Tree), are implemented. It is the point of entry for external players (e.g., service orchestrator, user CLI) that need to access NF's resources, modify (e.g., for configuration) or read NF parameters (e.g., reading statistics) and receive notifications from the NF fast path or slow path. Polycube defines a specific *control and management* module that performs the previously described functions. It exposes a set of `REST` APIs used to perform the typical CRUD (create-read-update-delete) operations on the NF itself; these APIs are automatically generated by the framework starting from the NF description (i.e., a YANG model of the NF), removing this additional implementation overhead to the programmer. To interact with the NF, an external player has to contact `polycubed`, which uses the service instance, contained in the URL, to identify which NF the request is directed and dispatches it to the corresponding NF control path, which in turn serves the request modifying its internal state or reflecting the changes to the NF data path instance. More details on how the control plane and the REST APIs of each NF are automatically generated is presented in Section VII.

3) Implementation: While the *fast path* is implemented by injecting the data plane programs in the eBPF kernel sandbox, both the *slow path* and the *control/mgmt plane* are implemented in user space, although in a different portion of the source code. In fact, the slow path is implemented by a callback named `packet_in`, which is fired only when the fast path detects an exception. Vice versa, functions in the control/mgmt plane are triggered by a different set of events, such as a request to the REST API of the NF (e.g., to read/write data in maps), expiring timers (e.g., for periodic cleanup tasks), and more.

From the performance point of view, the far majority of packets (actually, *all* packets in some NFs) are handled by the fast path, hence the userspace code does not introduce any performance penalty, nor it represents a performance-critical component for the run-time behavior or the NF.

V. Service Chaining Design

A Polycube service chain involves of a set of network function instances (i.e., Cubes) that are connected to each other by means of virtual ports, which are in turn peered with a Linux networking device or another in-kernel NF instance. In the standard model, eBPF programs do not have the concept of port from which traffic is received or sent out; it only provides a *tail call* mechanism to "jump" from one program to another. To provide this abstraction, Polycube uses a set of additional eBPF components and wrappers around the user-defined code; Figure 2 shows the resulting design.

When a packet traverses a chain in Polycube, it carries some metadata (e.g., ingress virtual port, module index), which are internally used by Polycube to correctly isolate the various NFs and implement the desired chain. In particular, the *cube index* is used to uniquely identify the Cube fast path inside the framework and it is uniquely generated when the Cube instance is created. Before injecting the Cube's fast path, Polycube augments the user-defined code with a set of *wrappers* that are executed *before* and *after* the NF itself. In particular, the *pre-processor* contains the set of functions necessary to process the incoming traffic, while the *post-processor* contains the helpers used by the fast path to send the traffic outside of the Cube.

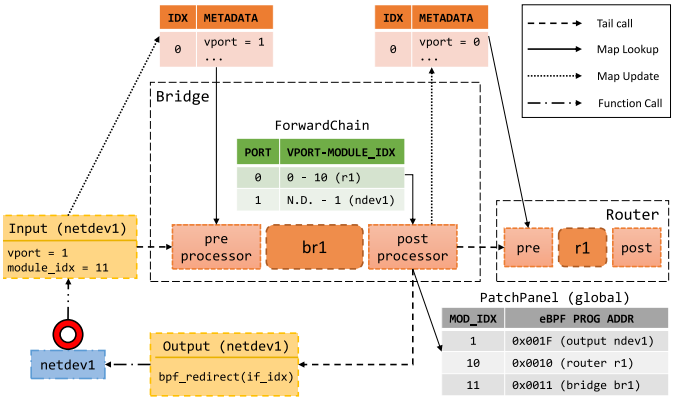


Fig. 2. Example of a service chain (bridge + router) in Polycube: internal details.

Pipeline Example: We will now walk through a simple example to illustrate how packets are passed through the Polycube service chain. In Figure 2 we have an instance of a Polycube Bridge NF, called `br1` with two virtual ports connected respectively to a Linux networking device (i.e., `netdev1`) and to the first port of an instance of *Router* Cube, called `r1`. First, when a new port is created in the `br1` instance, Polycube assigns a unique virtual port identifier (i.e., *vport*) to each port; in our case, the port attached to the router has id #0, while the other has id #1. At the same way, the router's port attached to the bridge has id #0.

When a new packet is received to the physical interface `netdev1`, it has to execute the `br1` fast path and be presented as coming from the virtual port #1. To support this abstraction, every time a Cube port is peered with a Linux networking device, Polycube loads two additional eBPF programs. The `Input` eBPF program, which is attached to the ingress hook of the interface, is loaded and compiled at runtime with some pre-defined information such as the *virtual port id* (i.e., #1 in the example in the figure) associated by Polycube to that Cube port and the `index` of the module to which the port is attached (i.e., #11). Upon the reception of a new packet from the physical interface attached to the bridge, the `input` program is triggered; it copies the *vport* value into an eBPF per-cpu array map shared with the bridge instance and performs a tail call to the `br1` pre-processor, using the hard-coded module index.[5]

At this point, the control passes to the *pre-processor* module of `br1`, which extracts the `vport` from the shared map (and possible additional metadata such as the packet length, headers) and invokes the `br1` code. The Cube fast path can then use the `vport` to send traffic outside as result of a forwarding decision; this is possible through a specific helper function contained in the *post-processor*, which will redirect the packet to the next module of the chain. The *post-processor* uses an additional auxiliary data structure, the `ForwardChain`, to obtain the index of the next module of the chain corresponding to the given vport. This map has a local scope and represents the actual connection matrix of the NF instance with the rest of the world. In our example, the lookup into the `ForwardChain` map with vport #0 returns the next virtual port id of `r1` (i.e., `0`) and the index of the eBPF program corresponding to that Cube (i.e., `10`). As before, the *post-processor* copies the vport into the shared map and "jumps" to the `r1` pre-processor. Once obtained the index of the next module, the post-processor performs a tail call using the `PatchPanel` to get the real address of the next eBPF program. On the other hand, if `br1` decides to redirect the packet to the port #0, the *post-processor* retrieves the next module index (i.e., `1`) from the `ForwardChain` and jumps to this module, which corresponds to the *Output* program associated with the physical interface. As for the *input* program, this module is injected with a pre-defined `if_index` of the netdevice, which uses in the `bpf_redirect()` helper function to send the packet out on `netdev1`.

To simplify the pipeline example, we have omitted the case in which the Cube is composed of several μCubes (Section IV-B1). Conceptually, the operations remain the same; the only difference is that, after the *pre-processor*, the code of the *MASTER* μCube is called, which in turn uses an internal *ForwardChain* to jump from one μCube to another.

VI. APIs and Abstractions

Polycube provides a set of high-level APIs and abstractions to simplify the development of a new NF, from both the control and data planes. For example, it adds useful abstractions to manage special packets, to cope with special processing that may complicate (and slow down) the fast path, or to react to special events such as timeouts. Table I shows some of the main helper functions introduced by Polycube at different levels of the NF code, i.e., the eBPF fast-path, the slow path and the control and management plane.

A. Transparent Port Handling

A Polycube NF instance is composed by a set of virtual ports that are uniquely identified through a *name* and an *index* inside the NF itself. Each port of the NF can be attached to a Linux network device or to another NF port by means of the *peer* parameter. When the fast path of the NF decides to redirect the packet to a specific output port it can use the `pcn_pkt_send()` function to send the packet to the next hop whether it is a net-device or another Polycube NF. Although the implementations for the above two types of next hops are quite different, Polycube hides this difference by providing a generic helper that receives the virtual index of the output port and, if the port is connected to a netdevice, redirects the packet to the attached netdevice, otherwise jumps directly to the next Polycube network function in the chain.

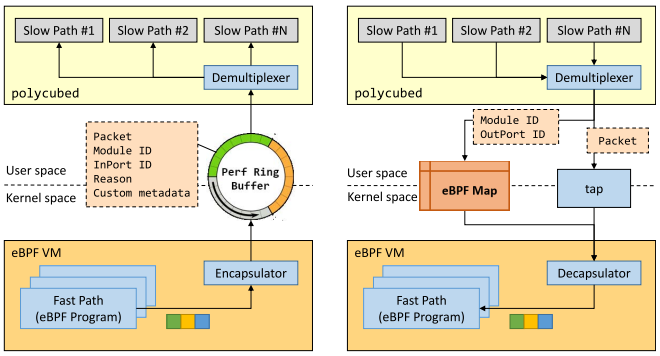
B. Fast-Slow Path Interaction

In Polycube, each instance of a NF has its own private copies of *fast* and *slow* paths; Polycube takes care of NF isolation by delivering packets generated by the fast path to the corresponding slow path instance and vice versa. It uses

[5]The only way to share information from one eBPF program to another is to copy the data into shared eBPF maps. For the internal communications, Polycube uses per-cpu maps, which provide better performance thanks to their lockless access. Since a packet is processed only within a single core (eBPF does not allow *preemption*), this mechanism is safe.

TABLE I
HELPER FUNCTIONS PROVIDED BY POLYCUBE AT DIFFERENT LEVEL OF THE NF

Level	Helper Function	Arguments	Description
Fast path (eBPF)	`pcn_pkt_send`	md, out_port	Redirect a pkt to a Cube interface (physical or virtual)
Fast path (eBPF)	`pcn_pkt_controller`	reason	Send a pkt to the slow path with a given reason
Fast path (eBPF)	`pcn_pkt_controller_md`	md, reason	Send a pkt to the slow path with a given reason and metadata
Fast path (eBPF)	`pcn_notify`	md, reason	Send a notification to the slow path with a reason and metadata
Fast path (eBPF)	`pcn_call_ingress_program`	index	Call the μCube at a given index attached to the ingress pipeline
Fast path (eBPF)	`pcn_call_egress_program`	index	Call the μCube at a given index attached to the egress pipeline
Slow path (user)	`pcn_pkt_in`	pkt, md, reason	Callback executed when a notification is sent to userspace
Slow path (user)	`pcn_send_packet_out`	pkt, dir	Send a packet out to the ingress or egress pipeline
Fast/slow path	`pcn_log`	level, txt	Print debug messages with a given verbosity level
Control plane	`pcn_reload`	code, idx	Reload the μCube at a given index with the new code
Control plane	`pcn_reload_all`	code[]	Reload all the μCubes of a given Cube with the new code



(a) Encapsulator (b) Decapsulator

Fig. 3. Message flow for Encapsulator and Decapsulator.

two separate (hidden to the developers) eBPF programs, the *Encapsulator* and *Decapsulator*, which are instantiated and injected into the kernel when `polycubed` is started.

a) *Encapsulator:* Figure 3(a) shows the flow of operations performed when sending a packet from the eBPF fast path to the slow path running in userspace. If the packet currently processed in the NF fast path requires additional inspections or further processing, it can be sent to the slow path module of the NF by means of the `pcn_pkt_controller()` helper. This function receives as parameters the reason why the packet has to be sent to the slow path and, optionally, additional meta-data fields. Polycube hides the implementation details of the communication between the eBPF fast path program and the NF slow path; it sends the packet to an eBPF control module (the *Encapsulator* shown in Figure 3(a)), that will copy the packet and its meta-data into a *perf ring buffer*, which is used by `polycubed` to read the corresponding data from userspace.[6,7] When a copy if the packet is sent to the slow path, the original one is retained in the fast path where it can continue the processing, or be dropped. Together with the custom metadata, the *Encapsulator* adds some internal information, such as the *index* of the Cube instance that has generated the message, which are used by the Polycube daemon to call the `pcn_pkt_in()` function of the associated NF's slow path.

b) *Decapsulator:* This component handles the reverse communication, which happens when the slow path (or the control plane) of the NF wants to inject a packet back in the fast path or send it out on a specific Cube port. In the first case, Polycube simulates the reception of the packet from a specific Cube port, which is specified by the control or slow path module through an apposite Polycube helper function, while in the latter case the output port of the Cube is provided. When called, this function triggers the execution of the `Demultiplexer`, which copies the *index* of the Cube originating the message into a specific eBPF map shared with the *Decapsulator*; then, it sends the packet on a `TAP` interface specifically created by `polycubed`. Differently from the `Encapsulator`, it does not use the *perf ring buffer* to communicate with the *Demultiplexer*, which is only available for the kernel-userspace communication and not for the opposite. The reception of a packet on the `TAP` interface triggers the execution of the *Decapsulator* eBPF program, which is attached to the eBPF hook point of the `TAP` interface. When executed, the *Decapsulator* extracts the index of the eBPF program to call from the map and jumps to the next program, following the same operations described in Section V.

C. Service Debugging

Debugging a NF that includes both data and control/management planes can be difficult because of the different context (kernel/userspace) the code is executed on. For instance, eBPF programs use the `bpf_trace_printk()` function to print debug messages; once the program is loaded, the verifier checks whether the program is calling this function and allocates additional buffers, which may slow down the processing of the function. Furthermore, message ordering

[6]eBPF provides specific helpers that allow to store custom data into a perf event ring buffer. Userspace programs can then use this buffer as a data channel for receiving events from the kernel.

[7]eBPF maps would be inefficient for this case because they should be sized with the largest packet length, even if in some cases we want to send a truncated packet to the slow-path.

can be reversed depending upon the context (user or kernel) the generating code was running on.

To solve the above problems, Polycube provides a debug helper that can be used to print debug messages in both data and slow/control planes. Polycube defines a new (and more efficient) `pcn_log()` helper that uses a *perf ring buffer* to send debug messages to `polycubed`, which redirects them to the current log file, as for the slow and control path. Finally, using different log levels, `polycubed` is able to dynamically remove all the references to the debug messages under the specified log level, reloading the NF fast path to reflect the changes.

D. Table Abstractions

To store the network function state across different runs of the same program or to pass configuration data from the control path to the fast path, a Polycube NF uses eBPF tables, which are defined into the NF fast path and are created when the program is loaded. Each eBPF table is associated to a scope that determines the possibility to read and/or modify the table content from another eBPF program. Polycube introduces the possibility to define `PRIVATE` tables, which are only accessible from the same μCube where they have been declared and `PUBLIC` tables, which are instead accessible from every μCube running in the machine. In addition, since Polycube supports the possibility to compose the network function data path as a collection of μCubes (i.e., simple eBPF programs), we added the concept of `SHARED` tables, where a table can in fact be shared between a given set of μCubes. In this case, when the table is instantiated, it is possible to specify the *namespace* within which this table will be shared.

E. Dynamic Fast Path Reloading

In addition to the capability of dynamically attach different Cubes to form a complex service chain, Polycube enables a single Cube to regenerate entirely its fast path (or part of it) by dynamically reloading the code of a μCube and inject it back into the kernel. This is particularly useful when the control plane of a Cube recognizes that a set of conditions cannot be met given the runtime configuration or traffic characteristics, or simply because it wants to update the code without causing any traffic disruption.

Once a Cube's control plane calls the `pcn_reload()` function (Table I), it provides the new source code and the index of the μCube that should be replaced. At this point, Polycube compiles and injects the new μCube, without attaching it to the Cube's fast path. When the new program is ready, first we attach the maps of the old instance to the new ones, then we atomically *swap* the code by replacing the pointer to the old program with the new one in the `PatchPanel` (Figure 2). This update is guaranteed to be not only atomic by the eBPF subsystem,[8] but also very fast, as it actually consists in updating a memory location. At this point, the new μCube will start processing the traffic and the old one is unloaded. The `pcn_reload_all()` performs the reloading on the whole set of μCubes composing the Cube's fast path; the reloading process is the same with the difference that, once the chain is ready the *MASTER* μCube is activated and attached to the Cube's *pre-processor*, as mentioned before.

[8]The `PatchPanel` map is *PROG_ARRAY* map, whose update is protected by the kernel RCU mechanism.

F. Support for Multiple Hook Points

Polycube supports two different type of NFs that correspond to the existing "attachments" points (a.k.a., eBPF hooks) available in the eBPF subsystem, namely Traffic Control (TC) and eXpress Data Path (XDP). The main difference between XDP and TC NFs are: *(i)* the initial context received as input by the triggered eBPF program, *(ii)* the type of standard eBPF helpers that the program is allowed to call and *(iii)* the return type of the program, which communicates the forwarding decision for that specific packet.

With respect to the initial context (which represents by far the most critical issue), TC cubes have access to the `sk_buff`, the standard Linux data structure for network packets, which is dynamically allocated upon packet arrival and it includes a set of metadata that result from the packet's parsing. Instead, XDP provides the `xdp_buff`, a raw buffer that contains only the received packet because it is allocated by the NIC driver before any TCP/IP stack processing. Although this contributes to a significant performance difference between the two hooks, on the other hand it provides to TC Cubes an additional level of information and stack's customization that cannot be done in XDP. For instance, TC programs can read or write the `sk_buff`'s internal values such as the *mark* value, used by a firewall to mark packets, the packet's *priority*, used to implement QoS, or to set the *queue_mapping* for the packet's transmission on a specific queue. However, this additional information and metadata available in the `sk_buff` complicates the re-writing of some packet values, which cannot be carried out by just modifying the packet data. In fact, the programmer has to take care of updating also the values in the `sk_buff` metadata, which is not required in case of XDP.

To simplify the development of data plane code, Polycube provides a set of hook-independent helpers that facilitate the update of many packet information (e.g., `pcn_vlan_push_tag()`). This is particularly useful to NF developers to provide the same version of a cube that works seamlessly, independently from the hook point at which it is attached to. In particular, Polycube "wraps" the execution of the program around two additional components that are executed *before* and *after* a packet enters and/or leaves the cube. Those wrappers take care of converting the original context into a standard Polycube format and perform the reverse translation on the opposite direction (Section V).

Of course, it is still possible to retrieve the original context (e.g., the `sk_buff` or `xdp_buff`) to apply hook-specific operations (e.g., set custom XDP metadata, update specific *skb* values). In this case, Polycube will allow the developers to separate the two versions of the code and compile the right one depending on the hook point selected when the cube is instantiated.

With respect to the remaining issues, Polycube cannot overcome the limitations of eBPF helpers that are not available in all hook points (e.g., `bpf_skb_get_xfrm_state()` available only in TC, or `bpf_redirect_map()`, available only in XDP); instead, it hides the different return values by defining new constants that are automatically mapped to correct ones when the program is instantiated (e.g., `RX_OK` maps onto `XDP_PASS`).

Also the interaction between the fast and slow path is different depending on the type of hook point in which the function is executed. As for the previous case, Polycube provides a transparent API for this interaction. An example is the *encapsulator* and *decapsulator* (Section VI-B), whose implementation depends on the hook point at which the service is attached to; when a packet from the control/slow path of a service is injected into the fast path, `polycubed` selects the appropriate control program depending on the type of the service instance.

As a concluding remark, XDP Cubes are particularly useful to implement NFs whose final action would result in packets redirected or dropped at this early stage (e.g., DDoS mitigation, load balancing) given the early stage in which packets are processed (*north-south* traffic). A TC Cube results more suitable to use in situation where we cannot avoid the `sk_buff` allocation. For instance, containers operate on virtual devices such as `veth` to send packets to their destination (likely, another container; *east/west* traffic). In this scenario in which the kernel operates only with the `sk_buff`, an XDP Cube will not bring any additional benefits. In addition, TC Cubes can leverage the TC *ingress* and *egress* eBPF hook, enabling the execution of two different piece of code for packets entering or exiting from a given interface. Vice versa, XDP supports only the *ingress* direction. Despite the above-mentioned differences, XDP and TC Cubes are mostly complementary to each other and can be sometimes used interchangeably or at the same time depending on the use case.

Finally, although eBPF supports application-level data processing (e.g., to operate on the traffic generated by a local database) with the SOCKET attaching point, Polycube supports only hooks that operate on *packets*, namely TC and XDP.

G. Support for Large Cube's Chains

While loops cannot be created within the eBPF code, they are still possible by performing the proper tail calls from one program to another. Hence, the eBPF subsystem enforces a hard limit of maximum 32 chained programs for a single packet, which translates into a limit of maximum 32 μCubes that can be called one after the other for a single packet. To support longer chains, Polycube keeps track of the number of hops (i.e., μCubes) traversed by the packet (using a similar mechanism as the one shown in Section V); once the 32 limit is hit, the packet is sent to a dedicated slow path in `polycubed` and then injected back into the kernel in the same point of the chain. In this way, the 32 limit will be reset and the packet can continue the normal processing. However, to avoid a possible performance degradation, this behavior is disabled by default.

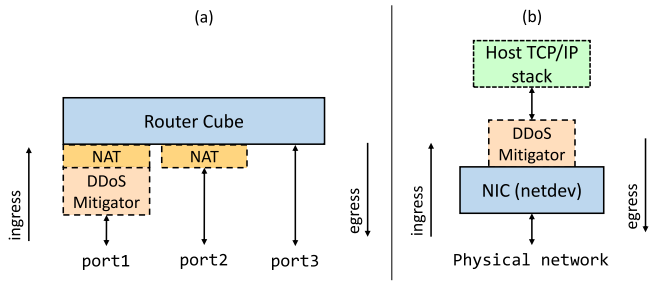


Fig. 4. (a) Transparent cubes attached to a port of the service. (b) Transparent cube attached to network device.

H. Transparent Services

When a packet is received on a specific port, a Polycube standard Cube can either (*i*) forward it to one of its output interfaces, (*ii*) redirect it to the slow path/control plane, (*iii*) return it to the Linux stack to continue its journey (the `RX_OK` action in Polycube), or (*iv*) drop it (`RX_DROP`). This process may depend on the specific port configuration and the behavior configured for each service. For example, a *router* Cube checks the routing table for each incoming packet, it determines the next-hop address and forwards the packet accordingly, using the port that connects to the next hop router. On the other hand, there are services whose duty consists simply in determining whether the packet has to continue its journey or not (i.e., either `RX_OK` or `RX_DROP`), possibly with updated protocol headers. Possible examples are firewalls (packets can either be forwarded or dropped), NATs, load balancers (packets have to be transformed according to specific rules), or even traffic monitors (no actions at all, just update counters).

While it seems that the above functions would also need to forward a packet on a specific port, we can note that the forwarding decision is always done through a logic that is orthogonal to the function itself; in fact, the forwarding can happen at L2 (e.g., using bridging rules), at network layer (e.g., using the IP routing mechanism), and more. Hence, having a fully functional firewall (or NAT, or load balancer) would imply integrating the forwarding logic into it, with all the associated variants (e.g., L2 or L3?), with a clear duplication of functions and additional complexity in terms of code maintenance.

To accommodate the above functions and to maintain an high degree of modularity, Polycube introduces the concept of *transparent cube*, i.e., a cube that cannot take any forwarding decision, hence cannot live alone and must be attached to either a port of a standard cube or to a Linux netdevice. Possible examples are depicted in Figure 4. The data plane of a transparent cube has only three allowed return values: (*i*) redirect the packet to the slow path/control plane, (*ii*) return it to the calling entity (e.g., a Linux netdevice, or a Polycube standard cube) (i.e., `RX_OK`), or (*iii*) drop it (i.e., `RX_DROP`).

By design (and for the sake of simplicity), a transparent cube can be attached to a single port (of a Cube/netdev); the case depicted in Figure 4a requires two NAT instances, each one operating on a single port of the router. Optionally, services can share some data, e.g., to coordinate the behavior of different instances (e.g., stateful filtering in firewalls).

A transparent cube is composed of an *ingress* pipeline that is called when a packet enters the service, and an *egress* pipeline that is called when the packet leaves the cube. In case the transparent service is attached to a netdev, the *ingress* pipeline is applied to incoming traffic, either in XDP or TC_INGRESS, while the *egress* pipeline is always executed in the TC_EGRESS hooks, given the unavailability of the egress hook in XDP. Transparent cubes can be stacked; in this case, they need to specify their position with respect to the others, hence defining the order of execution of the services. When attached, they have the possibility to inherit some specific parent's port configurations that can be used to automatically configure the service itself. For example, the NAT service attached to a router's port in Figure 4 can read the corresponding IP address and use it for the address translation; the same happens if the NAT is attached to a netdev, being the IP address associated to the above network port.

VII. Management and Control Plane

The capability to add (or remove) a network function dynamically (even from a remote server) into `polycubed` provides several advantages such as the possibility to update an existing NF, adding functionality without modifying the network functions currently deployed and running. To support this model, the Polycube core (i.e., `polycubed`) has been designed to be completely independent from the type of network function that is installed. `Polycubed` has no idea of how the network function is composed internally or what are its functionalities, and it takes only care of forwarding the request to the proper NF instance. This approach does not require changes to `polycubed` whenever changes to the individual NF are needed; when the NF is being updated, it is unplugged from the framework, updated and plugged-in again without affecting existing NFs. On the other hand, it complicates the NF design, which has to define the interface to the outside (i.e., the `REST` APIs). To simplify this process, Polycube uses `YANG` [48] models, each one describing a specific NF, to automatically synthesize the `REST` interface of the NF.

A. Model-Driven Service Abstraction

The `YANG` data modeling language allows to *(i)* model the structure of the data and the functionalities provided by the Polycube NF, *(ii)* define the semantic of the NF data and their relationship and *(iii)* express their syntax, which will be used to interact with the NF itself. When a new NF is registered, `polycubed` reads the provided `YANG` model and generates an internal representation of the NF data together with a specific path mapping table used to access those data from outside. Whenever a new request for that NF arrives, `polycubed` performs a validation of the input data (e.g., checking the correct format of an IP address, ports in a given range) according to the information specified in the `YANG` model, without having to rely on the NF itself for those "ancillary" tasks.

1) Service Base Data Model: Polycube provides to the developers a basic NF structure that can be extended to compose the desired network function, offering fundamental abstractions (e.g., VNF ports, port peers, hook type) that are used to simplify the interaction between the different NFs and system components. The basic structure, shown in the YANG Listing I, reflects the internal representation of a Polycube NF, with its primary parameters and components. Each NF within the framework is uniquely identified through a name, which is specified using the `service-name` extension; Polycube does not allow multiple NFs with the same name. The `service-min-kernel-version` is used to indicate the minimum kernel version required to execute the NF, since there are some eBPF functionalities that are available only on newer kernel versions; when the NF is loaded, Polycube checks if the host is running a kernel version greater than or equal to this value. Finally, the `service-description` and `service-version` are used to describe the current NF. While the previously mentioned information describe the NF itself (e.g., a firewall Polycube NF), the variables under the *grouping* statement are specific for each NF instance (e.g., a firewall *fw1*). Each instance is identified with a `name`, which is unique inside the NF scope, the `hook` point at which the instance is attached to, and a list of `ports`, identified with a unique *name* inside the NF instance and a *peer*.

2) Automatic REST API Generation: Polycube uses the information in the YANG model to automatically derive the set of REST APIs that are used to interact with the NF. Each YANG resource is automatically mapped to a specific URL, while the different HTTP methods are used to identify the operations required for a particular resource. The `GET` operation allows to obtain the current value of a given resource on the Polycube NF, the `POST` operation is used to create an instance of the resource, the `PATCH` operation is used to modify the current value of the resource and finally, the `DELETE` operation is used to delete the specified resource. This approach is similar to the one adopted by the RESTCONF specification [49], which aims at providing a programmatic interface for accessing data defined in YANG, allowing any client to communicate with the Polycube NF by just knowing its YANG module.

VIII. Implementation

A. Polycube Core

As of today, the code of Polycube, i.e., `polycubed` is implemented in 28k lines of C++ code, running within an unmodified Linux without having to install custom drivers or specific kernel modules. It only requires a v4.15 as minimum kernel version to run the daemon; then, each NF may have its own requirements depending on the functionalities that are used (e.g., eBPF helpers). The Polycube daemon contains both the code required to handle the different Polycube NFs but also the service-agnostic server proxies, which parses at runtime the YANG model of every loaded NF to generate the appropriate REST API and to perform the validation of the NF parameters within the server itself.

Polycube is built around the BPF Compiler Collection (BCC) [52], which provides a set of abstractions to interact with eBPF data structure or to load/unload eBPF programs, together with a compilation toolchain that include

TABLE II
A LIST OF NF IMPLEMENTED WITH POLYCUBE

Network Function	Minim. kernel	LoC FP	LoC SP	LoC CP (AU/MAN)	Description
Bridge	4.15	239	40	5798 / 1105	A L2 switch NF with support for VLAN and STP.
DDoS Mitigator	4.15	140	/	1850 / 20	A NF that drops (malicious) packets based on a blacklist applied on either IP src and dst addresses.
Firewall	4.19	1654	/	5951 / 2945	A firewall NF that drops or allows packets based on the configured rules.
Dynamic Monitor	4.15	/	/	2330 / 673	Generic NF used to inject eBPF code that monitors network traffic, collecting and exporting custom metrics. The data plane size depends on the injected code.
Load Balancer (DSR)	4.15	362	/	3476 / 566	A version of the Maglev scalable load-balancer [50].
Packet Capture	4.15	/	/	2511 / 822	A NF use to capture packets flowing through a Linux netdevice or between other cubes.
Policy-Based Forwarder	4.15	243	/	2605 / 240	A simple ACL-based forwarder.
Router	4.15	276	120	3168 / 1030	A router NF.
NAT	4.15	380	/	6302 / 208	A NF that supports Source, Masquerade, Destination NAT and Port Forwarding.
Iptables	5.3	2254	/	6409 / 1787	Special NF that emulates the behavior of iptables [51].
K8s Network Plugin	4.19	520	60	5010 / 156	Special NF that provides network connectivity within a k8s cluster (section IX-B4). The LoC CP indicate only the code needed to handle the NF, the interaction with the K8s components is not considered.

Clang/LLVM to allow a dynamic generation of the eBPF code that is injected in the kernel. Polycube extends those abstractions with additional helper functions targeted to networking services and to the Polycube NF structure. In particular, the availability of the compilation toolchain allows to re-compile the code at runtime, enabling more aggressive optimizations that can be dynamically applied within the NF. Although this approach, based on BCC, looks more limiting than other recent proposals such as BPF CO-RE (Compile Once—Run Everywhere), we can note that (*i*) the data structures available for TC and XDP programs are rather stable, different from the case of generic eBPF tracing programs that can attach to any point in the kernel, and (*ii*) such approaches can be supported in a future version of Polycube in which the compilation of the data plane program can be performed on a remote server.

B. Polycube Network Functions

To stress test the generality of the Polycube programming model and abstractions, we have implemented a large range of network functions. Table II briefly describes the type, role of each of them and the lines of code (LoC) required to implement the fast path (FP), the slow path (SP) and the control path (CP). In the former parameter we distinguish the LoC automatically generated from the YANG model (i.e., CP/AU) and the one manually written (i.e., CP/MAN).

At the time writing, there are 18 different NF implemented in Polycube, with an overall number of about 54k lines of code, which include both the C++/C code of the control and slow path and the eBPF code of the fast path. These implementations suggest that Polycube succeed in the role of providing a generic and highly customizable framework that can be used to implement a wide variety of NFs.

IX. EVALUATION

In this section we evaluate the performance of a set of Polycube NFs and we compare the results with existing in-kernel implementations. First, we measure the performance of standalone Polycube NFs and then we move to more complex scenarios where chains of NFs are involved (Section IX-B). Finally, we evaluate the overhead imposed by Polycube programming model when compared to baseline programs written using the vanilla eBPF (Section IX-C).

A. Setup

We run our experiments into a server equipped with an Intel Xeon Gold 5120 14-cores CPU @2.20GHz (hyper-threading disabled) 19.25 MB of L3 cache and two 32GB RAM modules. The packet generator is equipped with an Intel Xeon CPU E3-1245 v5 4-cores CPU @3.50GHz (8 cores with hyper-threading), 8MB of L3 cache and two 16GB RAM modules. We used Pktgen-DPDK [53] to generate 64-bytes UDP packets and to count the received packets. In fact, each server has a dual-port Intel XL710 40Gbps NIC, directly connected to the corresponding one of the other server. Both servers run Ubuntu 18.04.1 LTS, with the DUT running kernel v5.6 and the eBPF JIT flag enabled (the default behavior for newer kernels). For latency tests, we used Moongen [54], which generates the same traffic pattern as before but exploits the hardware timestamp of the NIC to determine the time a frame spends to return back to the sender; by default, one frame every millisecond is sampled. All tests were repeated ten times and the figures contain error bars representing the standard error calculated from the different runs.

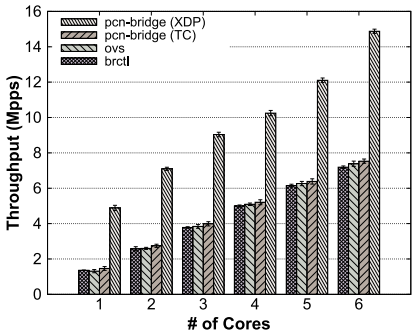


Fig. 5. Packet forwarding throughput comparison between Polycube *pcn-bridge* NF (in both XDP and TC mode) and "standard" Linux implementation such as Linux bridge (*btctl*) and OpenvSwitch (*ovs*).

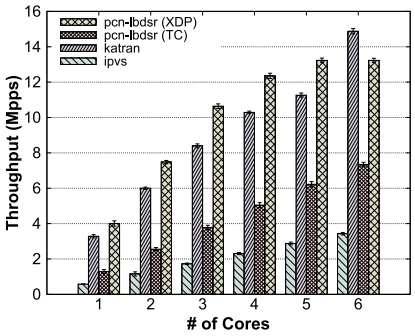


Fig. 6. Throughput performance between a Polycube load balancer NF (i.e., *pcn-lbdsr*), *ipvs*, the standard L4 load balancing software inside the Linux kernel and *Katran*, an XDP-based load balancer developed by Facebook.

B. Test Applications

1) Case Study 1 (L2 Switch): In this test scenario, we evaluate the performance of a Polycube NF that emulates the behavior of a fully functional L2 switch with support for VLAN and Spanning Tree Protocol (STP). The *pcn-bridge* data plane is implemented entirely in eBPF, including the MAC Learning phase. More complex functionalities such as the handling of STP protocol BPDUs or flooding, as results of a miss in the filtering database, are relegated to the slow-path, given the impossibility of performing such actions entirely in eBPF. We measure the UDP forwarding performance between *pcn-bridge* and commonly used L2 switch Linux tools such as Linux bridge (v1.5) and OpenvSwitch (v2.13).[9] In our test, *pcn-brigde* supports a maximum of 4K MAC entries, and we generate the traffic from 1K different source MAC towards 100 destination MAC. We start the test by sending traffic in both directions, in order to "warm-up" the MAC table, and then we measure the throughput on one direction. In this test, the slow path of *pcn-bridge* is involved when the packet has to be flooded, given the impossibility to clone and redirect a packet directly in the fast path towards multiple destination ports.[10] We can clearly notice, from Figure 5, that *pcn-bridge* outperforms the other tools in both TC and XDP mode, with a performance gain of about 3.6x for the latter. The advantages of XDP are more evident since packets are processed directly at driver level, avoiding the overhead given by the allocation of kernel data structures, which may be unnecessary for the simple forwarding use case. Moreover, we can notice how the performance of our *pcn-bridge* NF scale linearly with the number of cores used, reaching 10Gbps throughput with 64B packets with six cores involved.[11]

2) Case Study 2 (Load Balancer): In this test, we measure the performance of a Polycube load balancer NF (i.e., *pcn-lbdsr*). As for the previous scenario, this NF is implemented as a single μCube, with the data plane entirely handled in eBPF (no slow-path is involved). Polycube *pcn-lbdsr* can be configured with a list of virtual IPs (VIP), each one with an associated list of back-ends; we use the Maglev [50] hash to select the back-end server, which provides a better resilience to back-end server failures, a better distribution of the traffic load among the back-ends and the possibility to set different weights for each back-end server. To test this scenario, we used a fixed number of hosts,[12] as we compared the throughput results with IPVS v1.28 and Katran [1], an eBPF/XDP-based load-balancer developed and released as open-source by Facebook. Figure 6 shows the results of this test, with both *pcn-lbdsr* in TC and XDP mode that outperform *ipvs* by a factor of 2.2x and 6.5x respectively. Moreover, we want to notice that *pcn-lbdsr* in XDP mode offers performance comparable (or even higher) with *Katran*. This results is mainly given by the heavy use the Polycube NFs make of the dynamic reloading feature, which allows the NF to better adapt to the runtime configuration by compiling out features that are not required at runtime, hence, improving the overall data plane performance. The other big difference is that *Katran* is a standalone application built only for the load-balancing use case; on the other hand, Polycube offers a general framework to build and create complex NF chains, allowing to create more complex network typologies, while still providing performance comparable with "native" eBPF implementations.

3) Case Study 3 (Firewall): The Polycube *pcn-firewall* NF is implemented as a series a μCubes that compose the entire firewall pipeline (even in this case, the slow path is not involved). It implements the Linear Bit Vector Search (LBVS) [56] classification algorithm to filter packets, with a sequence of μCubes each one in charge of handling specific fields of the packet (e.g., IP source, destination, protocol, etc.).[13] Moreover, it is also able to "communicate" with the Linux routing table to check the next hop address before forwarding a packet to the egress interface.[14] For this test we used a synthetic ruleset generated by classbench [57], with

[9]All the tests with OpenvSwitch have been carried out on kernel v5.0 since it does not support earlier versions.

[10]This limitation holds only for the XDP version of *pcn-bridge*, while the TC version can exploit the `bpf_clone_redirect` helper to send a packets towards all the bridge's ports.

[11]To gradually use all cores, we have configured the hardware filtering on the NIC (i.e., Flow Director [55] rules) to redirect different flows to distinct NIC's RX queues.

[12]We used the same configuration of [38] for the load balancer use case, setting one virtual IP per CPU core and 100 back-end servers for each VIP.

[13]A more detailed explanation of the Polycube firewall architecture is provided in [51], whose data and control plane has been implemented as a standalone Polycube NF.

[14]eBPF provides a specific helper that can be used to lookup the FIB Linux table directly from the XDP code.

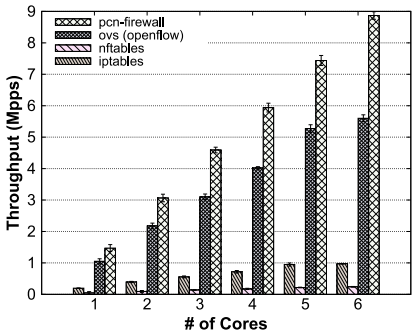


Fig. 7. Throughput performance comparing with 1000 rules between a Polycube firewall NF (i.e., *pcn-firewall*), *iptables* and *nftables*, which are two commonly used Linux firewalls and OpenvSwitch (*ovs*) with OpenFlow rules.

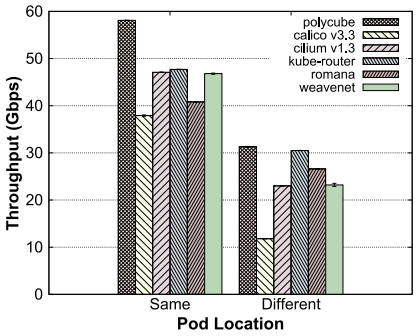


Fig. 8. Performance of different k8s network providers for direct Pod to Pod communication.

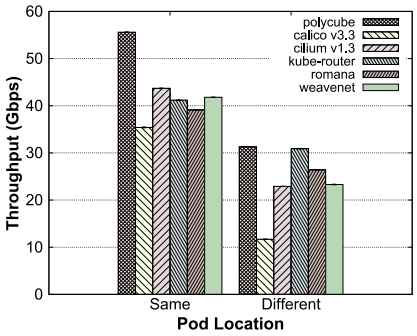


Fig. 9. Performance of different k8s network providers for Pod to ClusterIP communication.

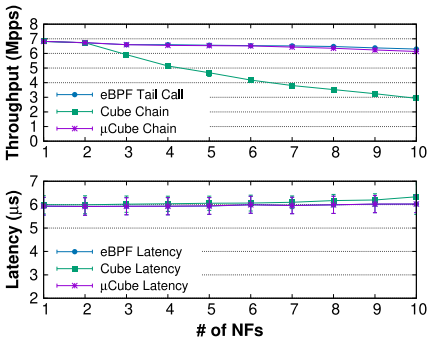


Fig. 10. Throughput and latency overhead of the Polycube service chain compared to "vanilla" eBPF.

all the rules matching the TCP/IP 5-tuple. The default rule of the firewall is to *drop* all the traffic, while only the matching flows are "allowed" and redirected to the second interface of the DUT. The generated traffic is uniformly distributed among all the rules so that all generated packets should be forwarded.

We compare the performance of *pcn-firewall* with both *iptables* and *nftables*, the most used packet filtering software used in the Linux subsystem today. Then, we also load the same ruleset as a set of OpenFlow rules in the OpenvSwitch pipeline[15] and we measure the performance under the same conditions mentioned before. The results are shown in Figure 7. Even in this case, we can clearly see how *pcn-firewall* outperforms the existing solutions by a 31.8x, 7.5x and 1.4x factor respectively for *nftables*, *iptables* and *ovs*. The reason of this is twofold. First, *pcn-firewall* implements a faster classification algorithm compare to the linear scanning implemented by Linux-native firewalls and second, it can adopt more aggressive optimizations thanks to the dynamic reloading feature of Polycube, allowing the control plane to specialize the packet processing behavior depending on the actual firewall configuration (e.g., the deployed ruleset).

4) Case Study 4 (K8s Network Provider): To demonstrate the capability of Polycube to enable the creation of complex applications created by chaining different network functions together, we present a real world use case that can be implemented within Polycube and the type of performance improvements that we can expect. In particular, we implemented a CNI plugin for Kubernetes [29], one of the most important open source orchestration system for containerized applications. A K8s network provider must implement Pod-to-Pod communication,[16] provide support for ClusterIP services[17] and security policies; our prototype supports all of them. Our overall design includes the five different components: a NAT NF (*pcn-nat*), a L3 routing module (*pcn-router*), a L2 switch application (*pcn-bridge*), a load balancer (*pcn-lb*) and a firewall component (*pcn-firewall*). These NFs are chained together by means of Polycube APIs, as shown in Figure 11, and are configured to support the main operations required by the k8s network plugin interface. Tests were carried out on a 3-node cluster, a master and two workers with Linux kernel v4.15, Intel Xeon CPU E3-1245v5 @3.50GHz with dual-port Intel XL710 40Gbps NIC cards connected point-to-point. We report the TCP throughput measured with `iperf3` using the default parameters; the server was always running in a Pod, while the client was either in a physical machine or in another Pod depending on the test.

In Figures 8 and 9 we asses the performance of our Polycube network provider compared to other existing solutions. In particular, we consider the Pod-to-Pod connectivity and the Pod-to-ClusterIP connectivity. Results show that the Polycube

[15]To generate the OpenFlow rules we used Classbench-ng [58].

[16]A Pod is the smallest manageable unit in a k8s cluster and is composed of a group of one or more containers sharing the same network.

[17]A ClusterIP is a type of service that is only accessible within a Kubernetes cluster through a *virtual* IP. When a Pod communicates with this *virtual* IP, the request can be mapped to an arbitrary Pod running within the same physical host or into another one.

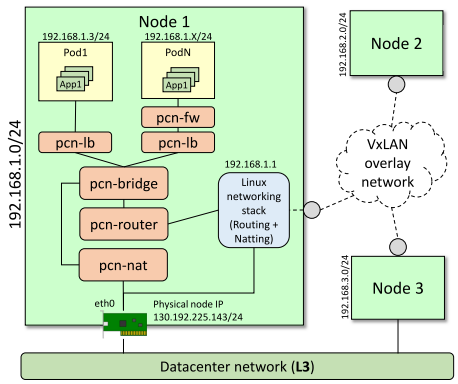


Fig. 11. Architecture of the Polycube K8s plugin.

TABLE III
COMPARISON BETWEEN VANILLA-EBPF APPLICATIONS AND A POLYCUBE NETWORK FUNCTION. ALL THROUGHPUT RESULTS ARE SINGLE-CORE

Application	Throughput (Mpps)	Latency/ Jitter (μs)	LoC (FP)	LoC (S/CP)
xdp_redirect	6.97	5.93 / 0.38	64	176
tc_redirect	1.60	6.16 / 0.25	53	56
pcn-simplefw (XDP)	6.86	6,00 / 0.39	17	0
pcn-simplefw (TC)	1.55	6.22 / 0.26	17	0

k8s plugin reaches 15-20% higher throughput than other solutions in the case server and client are on the same node. When pods are on different nodes, the advantage of the plugin becomes less evident because of the influence of the physical network, but still better than other solutions. Indeed, those providers often relies on existing kernel components, such as *iptables* and Linux bridge, that we proved in the previous case to be less efficient than the Polycube counterparts.

Although our k8s plugin achieves better performance than the others in the two cases under consideration, it is always comparable in terms of functionality with the existing solutions, which are both more stable and complete. The main purpose here is to demonstrate the generality of the Polycube programming model and the performance benefits that can be obtained from eBPF-based NFs.

C. Framework Overheads

In this section we evaluate the overheads imposed by Polycube programming model when compared to baseline eBPF programs written outside the Polycube environment.

1) Overhead for Simple NFs: To measure the baseline performance and the overhead introduced by the Polycube abstraction model to a single NF, we implemented the same operations performed by the `xdp_redirect` application [59], available under the Linux samples, as a standalone NF inside the Polycube framework (i.e., `pcn-simplefwd`). The application receives traffic from a given interface and, after swapping the source and destination L2 addresses of the packet, redirects it to a second interface.

Table III shows a comparison between two very simple vanilla eBPF applications and a Polycube NF that performs the same operations, attached to either XDP or Traffic Control (TC) hooks. As we can notice, Polycube introduces a very small overhead compared to vanilla eBPF applications, both in terms of throughput (6.86Mpps vs 6.97Mpps) and latency/jitter (6μs vs 5.93μs), which is required to provide the abstractions mentioned before (e.g., virtual ports). This is mainly given by the additional processing that happens before and after calling the fast path of the NF, which is totally hidden to the NF developer. As result, the number of LoC for both the fast-path (FP) and the slow and control path (S/CP) is considerably reduced, allowing the developer to focus on the core logic of the program and leaving the common tasks and the possible optimizations to the Polycube daemon. Note also that the sample vanilla applications that we are taking into account are extremely simple; for more complex applications, a developer using vanilla-eBPF has to implement, for example, the entire fast-slow path interaction, which requires a non-negligible amount of effort.

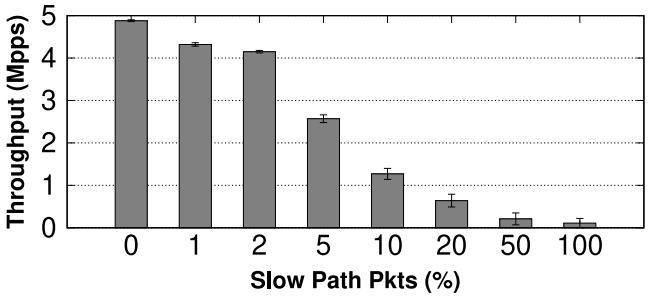


Fig. 12. Single core throughput of the *pcn-bridge* Cube with a different percentage of packets (64B) sent to the slow path.

2) Overhead for Chained NFs: Figures 10(a) and (b) shows the overhead introduced by the Polycube service chain in both throughput and latency, compared to the standard eBPF tail call mechanism. Of course, eBPF does not support any type of abstraction required by a NF framework; a tail call performs only an indirect jump from one eBPF program to another. On the other hand, Polycube uses a set of additional components and abstractions that are executed before a packet enters and leaves a NF, e.g., to ensure isolation or virtual port abstraction. This additional overhead is more evident when a packet runs through an increasing number of virtual Cubes but it is necessary to ensure the correct execution of the NF chain. On the other hand, if the same Cube is composed by a set of μCubes the overhead of crossing different μCubes is almost negligible, reflecting the same behavior of the tail call mechanism in the "standard" eBPF approach.

3) Slow Path Performance: To overcome the limitations of the eBPF sandbox, Polycube extends the in-kernel fast path with a slow path that can be used to perform operations on packets that cannot be done in the kernel (Section IV). In this subsection, we evaluate the performance of such component. In particular, we evaluated the *pcn-bridge* function showed before and we sent an increasing percentage of 64B packets to trigger the slow path processing (i.e., to perform flooding). The results showed in Figure 12 demonstrates that for higher rates the slow-path becomes a bottleneck. This overhead is

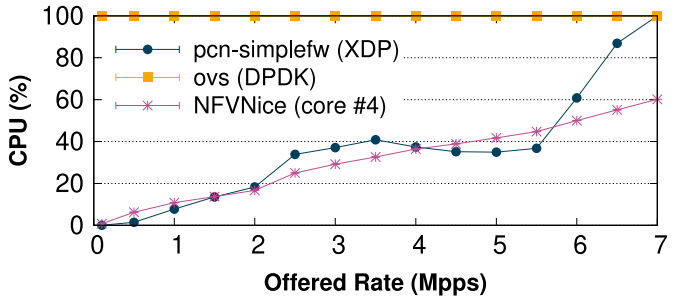


Fig. 13. Single core CPU usage comparison between *pcn-simplefw (XDP)* and *ovs (DPDK)*. The ovs (DPDK) line continues at 100% up until reaching maximum performance (i.e., 14.88Mpps with single OpenFlow *output* action. NFVNice plots only the 4th core, excluding the three ones running at 100%CPU; CPU consumption grows until 100% with 14Mpps.

mainly given by the double packet copy (and context switch) required to deliver the packet to the userspace application and send it back to the kernel, where it is injected into the Polycube service chain or to the output interface.

We note however that most of the applications implemented in Polycube do not require an high usage of the slow path, as shown in Table II, and for those where it is required, the percentage of slow path packets fall beyond the 5%, which is still acceptable. If a developer requires a higher slow path usage, other mechanisms should be considered such as `AF_XDP` [60], which however we do not cover in this article but it is part of our future work.

X. Additional Discussion

This section summarizes the most frequent discussions about Polycube, which are often due to the complexity of eBPF, a recent technology that is not yet fully known by the scientific community.

A. Polycube vs Other Userspace NFV Frameworks

The difference between kernel and user-space networking is well-known in the literature, with pro and cons on both sides that we partially explored in Section II. Høiland-Jørgensen *et al.* [38] have analyzed the performance differences between XDP and DPDK, showing a gap between the two approaches in favor of the latter; our tests simply confirm previous results. This overhead is almost inevitable, and is mainly given by the generality of the Linux operating system, which is structured in a way that make it easier to support different use cases and not only I/O intensive applications. However, it is important to note that for lower rates, Polycube provides better advantages in terms of CPU consumption compared to kernel bypass approaches. As example, Figure 13 compares the *pcn-simplefw* Cube against OvS-DPDK and NFVNice [61], a DPDK-based solution that provides fair scheduling of NFs without a continuous poll on the NIC. For NFVnice, we tested the *bridge* NF available in their repository [62], which actually looks a simple forwarding module. To run this application, NFVnice requires a total of 4 cores; three are used for master and TX/RX, plus another core to the actual NF, whose consumption increases with the traffic sent. Results confirm our initial statement that Polycube provides an excellent performance/efficiency trade-off compared to the existing solutions, particularly when assuming that the server workload does not include only NFV applications.

B. Reloading μCubes in Polycube

Polycube allows developers to reload the code of every single μCube inside the NF by using the `pcn_reload()` function (Section VI-E). This requires Polycube to re-compile the μCube's code by using the embedded Clang+LLVM toolchain, and inject the code in the kernel. The compilation time depends on the complexity of the code and the number of μCubes that have to be replaced. For the majority of NFs shown in Table II this time varies between 250 and 900ms. This value has to be added to the time required to inject the μCube in the kernel, which is about 1ms for a single μCube or 50ms for more complex Cubes such as *pcn-firewall*.

While the above control plane operations are in progress, the existing data plane pipeline continues to process packets, with an atomic swap of a function pointer performed only when the new pipeline is ready. To verify the impact of this operation, we measured the throughput that provides less than 1% of packet loss for the *pcn-simplefw* with different reload rates (i.e., starting from every 10ms), without noticing any change in the obtained numbers.

C. Scalability in Polycube

eBPF programs (and then μCubes) are executed on the same CPU core where the interrupt is dispatched, which means that the same Cube can be executed in parallel on different CPU cores, based on the outcome of the Linux load balancing interrupt algorithm. This behavior can be customized by either configuring the hardware RSS of the NIC or by using the `bpf_redirect_cpu()` helper to "redirect" a specific set of flows to a different CPU core, where the same chain of Cubes will be executed in parallel. Consequently, the scalability of Polycube functions is almost automatic, with no tasks left to the programmer (except using per-CPU data structures whenever possible, which avoid cache realignments between CPU cores).

D. Interactions Between Cubes and External Services

The interaction between Cubes and external services (e.g., to read the content of a database, or receive notifications from other application-level utilities) is devolved to the Cube's control plane, which can then "convert" the received information in data that can be pushed in the eBPF data plane (e.g., through *maps*). This limitation originates from the sandboxed nature of eBPF, which provides a very safe (but, in some cases, limiting) environment.

E. Isolation Between Cube's Chains

When instantiated, each Cube is associated with a different *namespace* that guarantees the proper resource isolation (e.g., eBPF maps in different cubes are not visible by default). This behavior can be modified when needed, allowing Cubes to share resources, such as a *connection tracking* table reused across different Cubes.

On the other hand, Polycube does not support the concept of having multiple tenants that are in charge of different NF chains. Although this behavior can be emulated with the *namespaces* mentioned before, multi-tenancy would require additional support inside Polycube to *split* traffic among the chains associated to different users. In this case, Polycube should instantiate an additional hidden program in front of the pipeline (following the same approach explained in Section V) that parses the packets and redirect them to the right tenant chains, as defined by high-level rules.

F. Polycube vs. Hardware Approaches (e.g., P4)

Polycube might benefit from the P4 language as an additional way to express the data plane semantic and operations performed when a packet is received. We explored this by extending the `p4c` compiler [63] to convert the P4 code into a Polycube-compatible format (using a polycube.p4 model), with the code available at [64]. This would first provide a "standard" way and semantic to express typical operations on a packet (e.g., parsing of headers, extraction of fields). The P4c-Polycube compiler can then implement common operations (e.g., packet parsing, header's extraction) in standard and more efficient way, avoiding potential performance differences that could come up with NFs written and deployed by different vendors.

XI. Limitations and Future Work

We now briefly discuss Polycube's main limitations and possible opportunities for future work.

Sharing μCubes across different NFs: Currently, Polycube does not allow to share a μCube among different NFs. A partial solution consists in duplicating the eBPF source code in both NFs, which results in the creation of two separated μCubes. However, this prevents a μCube with common operations (e.g., protocol headers parsing) to be shared at run-time among different NFs, hence potentially improving the efficiency of the entire eBPF service chain. A possible future work may consist in analyzing the chain of instantiated Polycube NFs and remove the redundant features at run-time, by re-injecting a customized and optimized version of the original Polycube service chain.

Scalability of Polycube's slow-path: Figure 3 showed the mechanism used to send packets from the (kernel) fast path to the (user-space) slow path for each running Cube. Since in the current prototype the `polycubed`'s *Demultiplexer* is a unique component shared among all the Cubes, it could become a potential bottleneck in case of slowpath-intensive services. This could be improved by instantiating a separate *perf ring* buffer for each Cube instance and use a separate thread to pull packets from it. However, the current solution is simpler and is appropriate in case services make infrequent use of slow path, as stated in Section IX-C3.

XII. Related Work

We elaborate in this section the related efforts beyond the work mentioned throughout this article.

Network Functions With eBPF: The first work that proposed the use of eBPF as subsystem to implement NF was presented in 2015, almost one year after the patch that added support for eBPF in the Linux kernel [65]. In their work, Sánchez and Brazewell [66] proposed an architecture for a "tethered" CPE with an implementation of the data plane entirely based on eBPF to obtain unprecedented efficiency, flexibility and portability across a wide range of hardware platforms. Following this approach, Ahmed *et al.* [67] proposed a network virtualization platform, called InKev, that uses eBPF as the base execution engine for building in-kernel virtualized network functions. InKev is probably the closest work to Polycube. Both works share the idea of using eBPF to build more complex data plane applications inside the Linux kernel but they different significantly in the design goals and details. Polycube aims at building a NF framework that fits better in the modern cloud era where loosely coupled, dynamically re-configurable and more specialized services are a must. Moreover, it focuses on building a well-defined communication interface for each services that is exposed externally, providing an automatic mechanism to generate such RESTful API that an orchestrator and other services can use for communication. Last but not least, Polycube provides the implementation of more than eleven different network functions that are ready to use and can be chained and configured properly to create custom and user-defined virtual network topologies. On the other hand, InKev does not provide such framework but only an draft implementation of the functionalities presented in this article.

Industry Efforts: Within industry, many companies started providing networking solutions based on eBPF. Facebook presented Katran [1] an XDP-based load-balancer deployed on Facebook's points of presence. Cloudfare also introduced L4Drop [68], an XDP DDoS Mitigation solution used in their system. Cilium [69] brings network security filtering mechanism to Linux container frameworks like Docker and Kubernetes; they use eBPF to enforce both network and application-layer security policies on the containers and pods. All of these solutions can be considered as separate implementations that serve for a specific purpose. On the other hand, Polycube offers an common software framework that addresses all the possible issues, limitations and optimizations that can be encountered within the eBPF subsystem. Moreover, it enable the interconnection of different services to create more complex applications and topologies, bringing the advantage of NFV to the world of *in-kernel* packet processing applications, something that, to the best of our knowledge, is not currently with any other open-source framework.

XIII. Conclusion

This article presents Polycube, a framework for developing, deploying and managing *in-kernel* virtual network functions. While most of the NFV framework today rely on kernel-bypass approaches, allowing userspace applications to directly access the underlying hardware resources, Polycube brings all the advantages and power of NFV to the work of in-kernel packet processing. It exploits the eBPF subsystem available in the Linux kernel to dynamically inject custom

user-defined applications into specific points of the Linux networking stack, providing an unprecedented level of flexibility and customization that would have been unthinkable before. In addition, Polycube has been created with in mind the new requirements brought by the microservice evolution in cloud computing. Polycube NFs follow the same continuous delivery development of server applications and being able to adapt to continuous configuration and topology changes at runtime, thanks to the possibility to dynamically inject and update existing NFs without any traffic disruption. At the same time, they offer a level of integration and co-existence with the "traditional" ecosystem that is difficult and inefficient to achieve with kernel-bypass solutions, enabling a high level of introspection and debugging that are fundamental in this environment.

We have implemented a vast range of applications with Polycube and shown that it is not only easy to deploy and to program but also improves network performance of existing in-kernel solutions. Polycube adds a very small overhead compare to vanilla eBPF applications but provides several abstractions that simplify the programming and deployment of new NFs and enables the creation of complex typologies by concatenating different NFs together, while maintaining and improving performance. We are continuing to explore the possible improvements and automatic optimizations that are possible within Polycube, some of which are currently rather primitive or specific for each application. Finally, we have also made Polycube available to the community at [34].

ACKNOWLEDGMENT

The authors would like to thank the many students and colleagues who collaborated to this project and contributed with ideas, code, comments, and in particular A. Palesandro, F. Parola, J. Pi and A. Shaikh with their help to move the project forward. The authors also thank VMware and FutureWei for their generous support.

REFERENCES

[1] C. Hopps. (Sep. 2019). *Katran: A High Performance Layer 4 Load Balancer*. [Online]. Available: xxhttps://github.com/facebookincubator/katran

[2] N. Katta *et al.*, "Clove: Congestion-aware load balancing at the virtual edge," in *Proc. 13th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2017, pp. 323–335. [Online]. Available: https://doi.org/10.1145/3143361.3143401

[3] K. He *et al.*, "PRESTO: Edge-based load balancing for fast datacenter networks," in *Proc. ACM Conf. Special Interest Group Data Commun. (SIGCOMM)*, 2015, pp. 465–478. [Online]. Available: https://doi.org/10.1145/2785956.2787507

[4] T. Barbette *et al.*, "A high-speed load-balancer design with guaranteed per-connection-consistency," in *Proc. 17th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Feb. 2020, pp. 667–683. [Online]. Available: https://www.usenix.org/conference/nsdi20/presentation/barbette

[5] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 503–514. [Online]. Available: https://doi.org/10.1145/2619239.2626316

[6] V. Jeyakumar, M. Alizadeh, D. Mazières, B. Prabhakar, A. Greenberg, and C. Kim, "EyeQ: Practical network performance isolation at the edge," in presented at the 10th USENIX Symp. Netw. Syst. Design Implement. (NSDI), 2013, pp. 297–311. [Online]. Available: https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/jeyakumar

[7] M. Nasimi, M. A. Habibi, B. Han, and H. D. Schotten, "Edge-assisted congestion control mechanism for 5G network using software-defined networking," in *Proc. 15th Int. Symp. Wireless Commun. Syst. (ISWCS)*, 2018, pp. 1–5.

[8] S. Miano, R. Doriguzzi-Corin, F. Risso, D. Siracusa, and R. Sommese, "Introducing smartNICs in server-based data plane processing: The DDoS mitigation use case," *IEEE Access*, vol. 7, pp. 107161–107170, 2019.

[9] G. Siracusano and R. Bifulco, "Is it a SmartNIC or a key-value store? both!" in *Proc. SIGCOMM Posters Demos (SIGCOMM)*, 2017, pp. 138–140. [Online]. Available: http://doi.acm.org/10.1145/3123878.3132014

[10] K. Lazri, A. Blin, J. Sopena, and G. Muller, "Toward an in-kernel high performance key-value store implementation," in *Proc. 38th Symp. Rel. Distrib. Syst. (SRDS)*, 2019, pp. 268–2680.

[11] Y. Le *et al.*, "UNO: Uniflying host and smart NIC offload for flexible packet processing," in *Proc. Symp. Cloud Comput. (SoCC)*, 2017, pp. 506–519. [Online]. Available: http://doi.acm.org/10.1145/3127479.3132252

[12] A. Bremler-Barr, Y. Harchol, and D. Hay, "OpenBox: A software-defined framework for developing, deploying, and managing network functions," in *Proc. ACM SIGCOMM Conf.*, 2016, pp. 511–524. [Online]. Available: https://doi.org/10.1145/2934872.2934875

[13] M. Kablan, A. Alsudais, E. Keller, and F. Le, "Stateless network functions: Breaking the tight coupling of state and processing," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Mar. 2017, pp. 97–112. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/kablan

[14] W. Zhang *et al.*, "OpenNetVM: A platform for high performance network service chains," in *Proc. Workshop Hot Topics Middleboxes Netw. Function Virtualization (HotMIddlebox)*, 2016, pp. 26–31. [Online]. Available: https://doi.org/10.1145/2940147.2940155

[15] G. P. Katsikas, T. Barbette, D. Kostić, R. Steinert, and G. Q. Maguire, Jr., "Metron: NFV service chains at the true speed of the underlying hardware," in *Proc. 15th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Apr. 2018, pp. 171–186. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/katsikas

[16] R. Laufer, M. Gallo, D. Perino, and A. Nandugudi, "CliMB: Enabling network function composition with click middleboxes," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 4, pp. 17–22, Dec. 2016. [Online]. Available: https://doi.org/10.1145/3027947.3027951

[17] S. Palkar *et al.*, "E2: A framework for NFV applications," in *Proc. 25th Symp. Oper. Syst. Principles (SOSP)*, 2015, pp. 121–136. [Online]. Available: https://doi.org/10.1145/2815400.2815423

[18] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. 12th USENIX Conf. Oper. Syst. Design Implement. (OSDI)*, 2016, pp. 203–216.

[19] DPDK. (Jun. 2018). *Data Plane Development Kit*. [Online]. Available: https://www.dpdk.org/

[20] L. Rizzo, "NetMap: A novel framework for fast packet I/O," in *Proc. 21st USENIX Security Symp. (USENIX Security)*, 2012, pp. 101–112.

[21] Cisco, "FD.io—Vector packet processing," Intel, San Jose, CA, USA, White Paper, 2017. [Online]. Available: https://fd.io/docs/whitepapers/FDioVPPwhitepaperJuly2017.pdf

[22] J. Martins *et al.*, "ClickOS and the art of network function virtualization," in *Proc. 11th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2014, pp. 459–473.

[23] M. Gallo and R. Laufer, "ClickNF: A modular stack for custom network functions," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, Jul. 2018, pp. 745–757. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/gallo

[24] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek, "The click modular router," *ACM Trans. Comput. Syst.*, vol. 18, no. 3, pp. 263–297, Aug. 2000. [Online]. Available: https://doi.org/10.1145/354871.354874

[25] L. Rizzo and G. Lettieri, "VALE, a switched Ethernet for virtual machines," in *Proc. 8th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2012, pp. 61–72. [Online]. Available: https://doi.org/10.1145/2413176.2413185

[26] I. Marinos, R. N. Watson, and M. Handley, "Network stack specialization for performance," in *Proc. ACM Conf. SIGCOMM*, 2014, pp. 175–186. [Online]. Available: https://doi.org/10.1145/2619239.2626311

[27] S. Han, K. Jang, K. Park, and S. Moon, "PacketShader: A GPU-accelerated software router," in *Proc. ACM SIGCOMM Conf.*, 2010, pp. 195–206. [Online]. Available: https://doi.org/10.1145/1851182.1851207

[28] A. Jaokar. (Mar. 2020). *An Introduction to Cloud Native Applications and Kubernetes*. [Online]. Available: https://web.archive.org/web/

20200413093940/https://www.datasciencecentral.com/profiles/blogs/an-introduction-to-cloud-native-applications-and-kubernetes
[29] G. Inc. (Jul. 22, 2019). *Kubernetes: Production-Grade Container Orchestration*. [Online]. Available: https://kubernetes.io/
[30] Netronome. (Jan. 2017). *Avoid Kernel-Bypass in Your Network Infrastructure*. [Online]. Available: https://web.archive.org/save/https://www.netronome.com/blog/avoid-kernel-bypass-in-your-network-infrastructure/
[31] E. Jeong *et al.*, "mTCP: A highly scalable user-level TCP stack for multicore systems," in *Proc. 11th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Apr. 2014, pp. 489–502. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/jeong
[32] K. Yasukata, M. Honda, D. Santry, and L. Eggert, "StackMap: Low-latency networking with the OS stack and dedicated NICs," in *Proc. USENIX Conf. Usenix Annu. Tech. Conf. (USENIX ATC)*, 2016, pp. 43–56.
[33] B. Pfaff *et al.*, "The design and implementation of OpenvSwitch," in *Proc. 12th USENIX Conf. Netw. Syst. Design Implement. (NSDI)*, 2015, pp. 117–130.
[34] P. Authors. (Jan. 2019). *Polycube: EBPF/XDP-Based Software Framework for Fast Network Services Running in the Linux Kernel*. Accessed: Oct. 25, 2020. [Online]. Available: https://polycube.network
[35] Cisco. (Jan. 2020). *Cloud-Native Network Function*. [Online]. Available: https://web.archive.org/web/20200414130638/https://www.cisco.com/c/dam/m/en_us/network-intelligence/service-provider/digital-transformation/knowledge-network-webinars/pdfs/1128_TECHAD_CKN_PDF.pdf
[36] Cilium authors. (Jan. 2019). *Diagram of Kubernetes/Kube-Proxy Iptables Rules Architecture*. [Online]. Available: https://web.archive.org/web/20200414131802/https://github.com/cilium/k8s-iptables-diagram
[37] M. Majkowski. (Jul. 2016). *Why We Use the Linux Kernel's TCP Stack*. [Online]. Available: https://web.archive.org/web/20200210223048/https://blog.cloudflare.com/why-we-use-the-linux-kernels-tcp-stack/
[38] T. Høiland-Jørgensen *et al.*, "The express data path: Fast programmable packet processing in the operating system kernel," in *Proc. 14th Int. Conf. Emerg. Netw. Exp. Technol. (CoNEXT)*, 2018, pp. 54–66. [Online]. Available: http://doi.acm.org/10.1145/3281411.3281443
[39] Cilium Authors. (Oct. 2020). *BPF and XDP Reference Guide*. [Online]. Available: https://docs.cilium.io/en/latest/bpf/
[40] Linux Programmer's Manual. (Aug. 2019). *BPF—Perform a Command on an Extended BPF Map or Program*. [Online]. Available: https://web.archive.org/web/20200428005645/http://man7.org/linux/man-pages/man2/bpf.2.html
[41] S. Miano, G. Retvari, F. Risso, A. W. Moore, and G. Antichi, "Automatic optimization of software data planes," in *Proc. ACM SIGCOMM Conf. Posters Demos (SIGCOMM)*, 2020. [Online]. Available: https://conferences.sigcomm.org/sigcomm/2020/cf-posters.html
[42] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, "A formally verified NAT," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 141–154. [Online]. Available: https://doi.org/10.1145/3098822.3098833
[43] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf. (SIGCOMM)*, 2016, pp. 314–327. [Online]. Available: https://doi.org/10.1145/2934872.2934881
[44] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, "Automated synthesis of adversarial workloads for network functions," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2018, pp. 372–385. [Online]. Available: https://doi.org/10.1145/3230543.3230573
[45] R. Iyer, L. Pedrosa, A. Zaostrovnykh, S. Pirelli, K. Argyraki, and G. Candea, "Performance contracts for software network functions," in *Proc. 16th {USENIX} Symp. Netw. Syst. Design Implement. (NSDI)*, 2019, pp. 517–530.
[46] F. Rath *et al.*, "SymPerf: Predicting network function performance," in *Proc. SIGCOMM Posters Demos (SIGCOMM)*, 2017, pp. 34–36. [Online]. Available: https://doi.org/10.1145/3123878.3131977
[47] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with EBPF: Experience and lessons learned," in *Proc. IEEE 19th Int. Conf. High Perform. Switch. Routing (HPSR)*, 2018, pp. 1–8.
[48] M. Bjorklund. (2016). *The YANG 1.1 Data Modeling Language*. [Online]. Available: https://tools.ietf.org/html/rfc7950
[49] A. Bierman, M. Björklund, and K. Watsen, "RESTCONF protocol," IETF, RFC 8040, Jan. 2017. [Online]. Available: https://rfc-editor.org/rfc/rfc8040.txt
[50] D. E. Eisenbud *et al.*, "Maglev: A fast and reliable software network load balancer," in *Proc. 13th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Santa Clara, CA, USA, 2016, pp. 523–535. [Online]. Available: https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/eisenbud
[51] S. Miano, M. Bertrone, F. Risso, M. V. Bernal, Y. Lu, and J. Pi, "Securing linux with a faster and scalable iptables," *SIGCOMM Comput. Commun. Rev.*, vol. 49, no. 3, pp. 2–17, Nov. 2019. [Online]. Available: https://doi.org/10.1145/3371927.3371929
[52] BCC Authors. *BPF Compiler Collection (BCC)*. Accessed: Oct. 20, 2020. [Online]. Available: https://web.archive.org/web/20181106133143/https://www.iovisor.org/technology/bcc
[53] DPDK. (Aug. 2018). *Pktgen Traffic Generator Using DPDK*. [Online]. Available: http://dpdk.org/git/apps/pktgen-dpdk
[54] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, "Moongen: A scriptable high-speed packet generator," in *Proc. Internet Meas. Conf.*, 2015, pp. 275–287.
[55] Intel. *Flow Director and Memcached Performance*. Accessed: Oct. 20, 2020. [Online]. Available: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/intel-ethernet-flow-director.pdf
[56] T. Lakshman and D. Stiliadis, "High-speed policy-based packet forwarding using efficient multi-dimensional range matching," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 28, no. 4, pp. 203–214, 1998.
[57] D. E. Taylor and J. S. Turner, "ClassBench: A packet classification benchmark," *IEEE/ACM Trans. Netw.*, vol. 15, no. 3, pp. 499–511, Jun. 2007.
[58] J. Matousek, G. Antichi, A. Lucansky, A. W. Moore, and J. Korenek, "ClassBench-NG: Recasting ClassBench after a decade of network evolution," in *Proc. ACM/IEEE Symp. Architect. Netw. Commun. Syst. (ANCS)*, 2017, pp. 204–216.
[59] L. Community. (May 2020). *XDP_Redirect_Map_Kern Application*. [Online]. Available: https://web.archive.org/web/20200522140154/https://github.com/torvalds/linux/blob/master/samples/bpf/xdp_redirect_map_kern.c/
[60] M. Karlsson and B. Töpel, "The path to DPDK speeds for AF XDP," in *Proc. Tech. Conf. Linux Netw. (Netdev)*, 2018. [Online]. Available: https://www.semanticscholar.org/paper/The-Path-to-DPDK-Speeds-for-AF_XDP-Karlsson/92abbc6c959f5ef71ad51a154ac8954995308712
[61] S. G. Kulkarni *et al.*, "NFVnice: Dynamic backpressure and scheduling for NFV service chains," in *Proc. Conf. ACM Special Interest Group Data Commun. (SIGCOMM)*, 2017, pp. 71–84. [Online]. Available: https://doi.org/10.1145/3098822.3098828
[62] OpenNetVM authors. (Oct. 2020). *NFVnice Sample Bridge Application*. [Online]. Available: https://github.com/sdnfv/openNetVM/tree/experimental/nfvnice-reinforce/examples/bridge
[63] P. L. Consortium. (Oct. 2020). *p4Lang-P4C: P4 Reference Compiler*. [Online]. Available: https://github.com/p4lang/p4c
[64] R. Marchi. *Polycube Extention of the P4C Compiler*. Accessed: Oct. 2020. [Online]. Available: https://github.com/richiMarchi/p4c/tree/polycube_translation
[65] A. Starovoitov. (Mar. 2014). *Net: Filter: Rework/Optimize Internal BPF Interpreter's Instruction Set*. [Online]. Available: https://patchwork.ozlabs.org/patch/333456/
[66] F. Sánchez and D. Brazewell, "Tethered linux CPE for IP service delivery," in *Proc. 1st IEEE Conf. Netw. Softw. (NetSoft)*, 2015, pp. 1–9.
[67] Z. Ahmed, M. H. Alizai, and A. A. Syed, "InKeV: In-kernel distributed network virtualization for DCN," *SIGCOMM Comput. Commun. Rev.*, vol. 46, no. 3, pp. 1–6, Jul. 2018. [Online]. Available: https://doi.org/10.1145/3243157.3243161
[68] A. Fabre. *L4Drop: XDP DDoS Mitigations*. Accessed: Oct. 20, 2020. [Online]. Available: https://web.archive.org/web/20190927231336/
[69] Cilium Authors. (2020). *Cilium: API-Aware Networking and Security Using eBPF and XDP*. [Online]. Available: https://github.com/cilium/cilium

Sebastiano Miano (Member, IEEE) received the master's degree in computer engineering from the Politecnico di Torino, Italy, in 2015, where he is currently pursuing the Ph.D. degree. During his research career, he spent several months in both industrial and academic institutions such as EIT Digital, San Francisco, CA, USA, and the University of Cambridge, Cambridge, U.K. His research interests include programmable data planes, software defined networking, and high-speed network function. virtualizations.

Fulvio Risso (Member, IEEE) received the M.Sc. and Ph.D. degrees in computer engineering from Politecnico di Torino, Italy, in 1995 and 2000, respectively. He is currently an Associate Professor with Politecnico di Torino. He has coauthored more than 100 scientific papers. His research interests focus on high-speed and flexible network processing, edge/fog computing, software-defined networks, and network functions virtualization.

Mauricio Vásquez Bernal received the M.Sc. degree in computer engineering from Politecnico di Torino, Italy, in 2015, where he collaborated as a Research Assistant. He is interested in computer networks, cloud computing technologies, tracing, and monitoring of distributed systems.

Matteo Bertrone received the M.Sc. degree in computer engineering from the Politecnico di Torino in 2016, where he collaborated as a Research Assistant. He is particularly interested in cloud native computing, with a focus on monitoring and security. His research interests include software-defined networking, high-performance networking, and data-plane programmability.

Yunsong Lu is the Board Chair of Linux Foundation IO Visor Project and the Co-Founder of Polycube Project. He was a Visiting Scholar with Stanford University. He co-founded Amiasys Corporation, where he is currently serves as a CEO. Before Amiasys, he founded Virtual Networking Lab, Huawei and created the elastic virtual switch, software-defined network acceleration, AI-driving networking (network twins) products and technologies as the fundamental and key building blocks of SDN, and cloud and edge networking. He worked with Sun Microsystems and Oracle, where he started his journey of pursuing and creating networking technologies.