Performance Monitoring with H^2: Hybrid Kernel/eBPF data plane for SRv6 based Hybrid SDN

Andrea Mayer[a,b], Pierpaolo Loreti[a,*], Lorenzo Bracciale[a], Paolo Lungaroni[b], Stefano Salsano[a,b], Clarence Filsfils[c]

Revision R2 - October 2020 - Accepted in Elsevier Computer Networks, Special Issue on Challenges and Solutions for hybrid SDN

[a]*Department of Electronic Engineering, University of Rome "Tor Vergata", Rome, Italy*
[b]*CNIT, Rome, Italy*
[c]*CISCO Systems, Brussels, Belgium*

Abstract

Segment Routing with IPv6 (SRv6) is a leading Hybrid SDN (HSDN) architecture, as it fully exploits standard IP routing and forwarding both in the control plane and in the data plane. In this paper we design and evaluate an efficient Linux software router in an HSDN/SRv6 architecture based on a programmable data plane architecture called HIKE, which stands for HybrId Kernel/eBPF forwarding. HIKE architecture integrates the conventional Linux kernel packet forwarding with custom designed eBPF/XDP (extended Berkeley Packet Filter/eXtreme Data Path) bypass to speed up performance of SRv6 software routers. Thus, in addition to the hybrid SDN based forwarding, we foster an additional hybrid approach inside a forwarding engine, taking the best from both worlds. Therefore, considering the two different conceptual levels of hybridization, we call our overall solution *Hybrid squared* or H^2.

We have applied the H^2 solution to Performance Monitoring (PM) in Hybrid SDNs, and we show how our HIKE data plane architecture supports SRv6 networking and

---

*Corresponding author

Email addresses: `andrea.mayer@uniroma2.it` (Andrea Mayer), `pierpaolo.@uniroma2.it` (Pierpaolo Loreti), `lorenzo.bracciale@uniroma2.it` (Lorenzo Bracciale), `paolo.lungaroni@cnit.it` (Paolo Lungaroni), `stefano.salsano@uniroma2.it` (Stefano Salsano), `cfilsfil@cisco.com` (Clarence Filsfils)

Performance Monitoring (in particular Loss Monitoring) allowing a significant increase in performance: implementation results show a remarkable throughput improvement of five times with respect to a conventional Linux based solution.
Keywords: Linux Networking, eBPF, XDP, SRv6, Performance Monitoring, Loss Monitoring

1. Introduction

Segment Routing (SR) [1, 2] is one of the prominent options for bringing Software Defined Networking (SDN) into Service Providers' Networks, going beyond its application into Data Center Networks. According to a vendor's statement, SR is the "defacto SDN Architecture" [3]. SR architecture has been implemented with the MPLS data plane (SR-MPLS) [4] and with the IPv6 data plane (SRv6) [5]. In this work, we will consider SRv6, the most recent of the two solutions. SRv6 is attracting a lot of interest from industry and academia and there are several on-field deployments [6].

SRv6 coupled with SDN is actually a Hybrid SDN (HSDN) solution, because it fully exploits standard IP routing and forwarding both in the control plane and in the data plane. The key idea of Segment Routing is that a *network program* can be added in the packet headers at the edge of an SR domain. In SR architecture, the *network program* corresponds to an *SR policy*, i.e. a list of *Segments* represented by their identifiers (*SID - Segment ID*). In SRv6, the SIDs are IPv6 addresses. An SR Policy is represented by a Segment List, which is a list of IPv6 addresses. For most of the use cases, the SDN controller can interact only with the nodes at the network edge (*Edge Routers*), by instructing them to inject the proper Segment Lists into packet headers. This approach results in a highly scalable HSDN architecture, that will be referred to as *HSDN/SRv6*: the network core is mostly operating with traditional IP routing protocols and there is no need for the network core nodes to interact with the SDN controller to configure single services. The amount of state information that needs to be maintained in the core is dramatically reduced with respect to a traditional Openflow based SDN, in which the SDN controller needs to install flows also in the core nodes.

It is worth noting that the so called SD-WAN solutions [7] share the same philosophy as HSDN/SRv6. Also, in SD-WANs the controller interacts only with the network edge and controls the network services from the edge. The key difference and advantage of HSDN/SRv6 architecture vs. legacy SD-WANs is that using SRv6 it is also possible to interact with core networks and influence the processing of packets in the core network. In other words, legacy SD-WANs are pure *overlay* networking solutions, while SRv6 offers the possibility to influence both *overlay* and *underlay* networking. A typical example is using HSDN/SRv6 architecture to implement VPN services with Service Level Agreements (SLAs) [8].

In this paper we consider a Linux software router in an HSDN/SRv6 architecture exploiting the extended Berkeley Packet Filter (eBPF) [9] and eXpress Data Path (XDP)[10]. We propose a data plane architecture called HIKE, which stands for HybrId Kernel/eBPF forwarding. The HIKE architecture highlights the need for integrating the packet forwarding and processing based on a "normal" Linux kernel with the ones based on custom designed eBPF programs. Thus, in addition to the hybrid approach between SDN based forwarding vs. legacy IP based forwarding, we envisage an additional "hybrid" approach "inside" a forwarding engine, i.e. based on a flexible and modular combination of the Linux kernel and a eBPF/XDP bypass. Therefore, considering the two different conceptual levels of hybridization, we shall call our overall solution *Hybrid squared* or H^2. HSDN networks are designed to reduce the amount of state information in the network nodes and the amount of interactions between nodes and the controller, compared to a pure SDN network; a challenge of the H^2 architecture is to further limit the amount of state information of a hybrid IP/SDN network and the interactions with the SDN controller in order to improve scalability and to be able to offer complex services such as SFC,Tunneling/VPN, Monitoring, etc. Moreover we aim at devising an architecture that takes advantage of the efficiency of eBPF but at the same time is modular and flexible. eBPF processing is very fast for simple operations comprising few memory accesses, but when eBPF is used for more complex operations and in a modular way (i.e. being able supporting multiple different services in parallel), the solution is no longer so straightforward. As an important field of application for the H^2 solution, we focus on Performance Monitoring (PM) in Hybrid SDNs. We consider the need to implement accurate and real-time per-flow monitoring of QoS parameters (delay, jitter, loss). The implementation of accurate per-flow monitoring may severely impact the performance of the data plane and thus call for an efficient implementation. We show how our HIKE data plane architecture supports Performance Monitoring (in particular Loss Monitoring) with a very limited impact on performance.

The main contributions of this work are:

- to present of our vision of SRv6 as a key component of an Hybrid solution based on IP routing and SDN control;
- to devise an Hybrid solution based on eBPF/XDP and traditional kernel forwarding in the forwarding plane on a Linux router;
- to integrate the two hybrid solutions in a flexible architecture;
- to implement and evaluate the proposed solution considering the performance monitoring scenario.

The paper is organized as follow: we first introduce the hybrid SDN model based on the SRv6 technology (HSDN/SRv6) from an system architecture viewpoint using the networks Performance Monitoring as a reference case. Then in section 3, we introduce the Linux data plane for HSDN/SRv6, describing both the Kernel based forwarding and the eBPF/XDP platform. In section 4, we present HIKE, our proposed hybrid data plane architecture. An open source proof-of-concept of HIKE has been developed, the upper bounds of its performance have been measured, as described in section 5. Section 6 analyses the related work. Finally, we draw some conclusions in section 7.

2. SRv6 as HSDN Solution

In this section we are going to briefly introduced SRv6 technology detailing how it is used in an HSDN architecture.

2.1. HSDN Network Scenario

Let us consider the reference network depicted in Figure 1. Node A and Node B represent respectively the ingress and the egress node to an SRv6 domain. At the edge

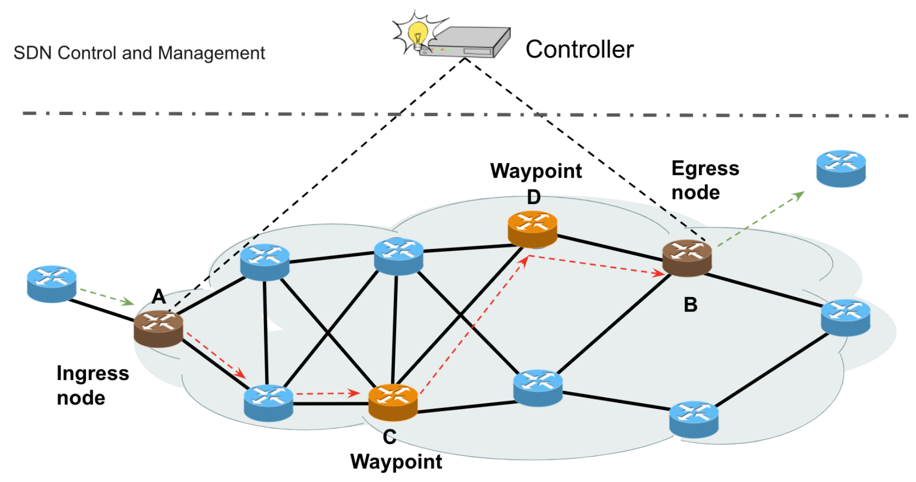


Figure 1: Reference SRv6 network scenario

of the network the packets are encapsulated in SRv6 traffic streams, i.e. the ingress node adds an outer IPv6 header to the packets that can include a Segment Routing Header (SRH). In SRv6 terminology, an *SR policy* is applied to the incoming packets. The SR policy corresponds to a *SID List* that is included in the new added SRH of the outer IPv6 packet header. In this example, the first two SIDs (IPv6 addresses) included in the list represent the two SRv6 waypoints (C and D) that the packet must pass through the network. The packet forwarding between the waypoints is operated using the standard IP protocol. Therefore, the intermediate nodes between two waypoints may even be plain IPv6 routers that do not support SRv6. When the packet arrives to the egress edge node, the inner IP packet is decapsulated, i.e. the outer IPv6 header and SRH are removed.

2.2. SRv6 Networking programming and HSDN

An important concept of SRv6 is network programming. The SRv6 vision defines packet processing in network nodes as the execution of a program, whose *instructions* are represented by the SIDs inserted by the ingress node in the Segment List in the SRH. Thus SIDs can be used not only as standard IP destination addresses (i.e. as *topological* instructions) but also as commands to execute functions in the different

SRv6 nodes of the network (i.e. as *service* instructions, that can range from simple operations on packet headers to arbitrarily complex behaviors. In [11] a set of instruction types or *behaviors* are defined, for example to specify a path through the network for each traffic flow/packet (e.g. 'End' or 'End.X') or to create VPNs (e.g. 'End.DX2' or 'End.DX2V'), etc.

Network programming allows an additional level of flexibility for the HSDN network. Indeed in SRv6 networks it is not necessary to control all the nodes, but often it is enough to control the node that operates the encapsulation to activate functions along the path of the packet. Thus if it is necessary to measure performances of a flow or even of a single packet, the controller can interact with the ingress node inserting in the SID list all operations to carry out in the different nodes along the network path.

Network programming technology allows for greater scalability of the HSDN by reducing the state information stored in the network. It also allows a high degree of granularity in the application of the functions that is difficult to achieve with standard controllers.

2.3. SRv6 Southbound Interface

To date, no standard Southbound Interface (SBI) has been defined for SRv6 HSDN networks. Few proposals for the SBI are available in the literature and the most complete is presented in [12]. In [12], the authors discuss the actual needs of SRv6 nodes and also compare them with SDN enabled routers that support the traditional Openflow interface. Focusing on Performance Monitoring, in [13] the authors present a HSDN/SRv6 architecture in which the SDN controller is able to manage per-flow Loss Monitoring of the SRv6 infrastructure. Our work is based on the Performance Monitoring architecture described in [13].

2.4. Performance Monitoring in HSDN/SRv6 networks

Performance Monitoring (PM) standardization activities for SRv6 are still ongoing, and at the moment the solution that seems to have reached a higher degree of maturity is one based on the TWAMP light protocol [14]. This technique can be an effective solution to enable the monitoring of some performance metrics such as loss and one-way or two-way delays following the so called fate sharing paradigm, according to which probe and data packets share the same network “fate”. Data collection takes place with test UDP packets transmitted on the same measured path. The test UDP packets collect the one way or two way PM data and make them available to the node which requested the measurement. Moreover, the TWAMP based PM supports the Alternate-Marking Method for accurate loss monitoring that can be applied to both IPv6 and SRv6 flows as specified in [15] and presented by [16].

3. Linux SRv6 Networking and the eBPF/XDP fast path

In this section, we first introduce the envisage application scenarios of Linux HSDN/SRv6 enabled routers and then we presents some details of SRv6 networking implementation approaches focusing on the eXpress Data Path (XDP).

3.1. Linux HSDN/SRv6 Applications

Taking into account the reference scenario described in section 2.1, there are several deployments in which HSDN/SRv6 enabled routers can be implemented using Linux, especially at the network edge. The most prominent example is an SRv6 powered Data Center infrastructure. In such infrastructure, a typical scenario is to have software based (Linux) routers running in the servers of the data center, playing the role of SRv6 Edge Routers, while the internal nodes are hardware based switches/routers. Likewise, with NFV (Network Function Virtualization) it is very common to have Linux based virtual appliances. A third important use case for Linux based implementations is represented by the so called “white box” networking devices. These devices are based on off-the-shelf computing and networking hardware, powered by an open source operating system (typically Linux based).

Let us consider the first example described above, i.e. a data center server also playing the role of a HSDN/SRv6 Edge Router. It is feasible and efficient to implement networking operations such as encapsulation, decapsulation, and performance measurements in such servers rather than in conventional routers. In fact, server running Linux can support all the traffic handling operations needed by HSDN/SRv6 networks, either natively at kernel level, or using additional frameworks like for example Vector Packet Processing [17] (VPP). It is even possible to enhance the Linux based processing by exploiting hardware accelerated networking cards, e.g. using the Data Plane Development Kit (DPDK) libraries.

3.2. Implementation aspects of Linux SRv6 Networking

In order to support the needed HSDN/SRv6 functionality in a Linux router, it is possible to use several different approaches. We list some of these approaches hereafter:

- SRv6 default implementation in Linux kernel networking, based on the "Lightweight Tunnels" (LWT).
- SRv6 implementation in the Vector Packet Processing (VPP) framework that is capable to exploit DPDK hardware acceleration.
- eBPF/XDP framework.

VPP is a kernel bypass solution which is intended to process packets in userspace. By giving full control of the NIC to an user-space program, the kernel overhead is reduced and it is relevant enough when working at speeds of 10Gbps or higher. However, userspace networking (such as VPP) comes with several disadvantages. The most significant one (considering this work) is that the kernel-space is completely skipped. As a consequence, all the networking functionalities provided by the kernel are skipped too. VPP networking programs operate like sandboxes, which limit their ability to interact and be integrated with other parts of the OS. Therefore, it is very difficult to design real hybridization between VPP like solutions and the Linux kernel networking stack. Thus, we focus on the first (Linux kernel) and on the third (eBPF/XDP) approaches and we propose an hybrid approach among these two. We point out that the eBPF/XDP is actually integrated in the Linux kernel, but for simplicity we will use "*normal* kernel" to refer to the approach that does not use eBPF/XDP. eBPF/XDP is extremely good for performance reasons, but we can identify some critical shortcomings in its utilization. eBPF/XDP can be used to execute specific tasks, by having the eBPF *Virtual Machine* process the packets. At the time of writing, it is rather difficult to combine multiple operations to be executed with eBPF/XDP in a modular way. It is also difficult to use information that it is dynamically updated by the control plane inside an eBPF program.

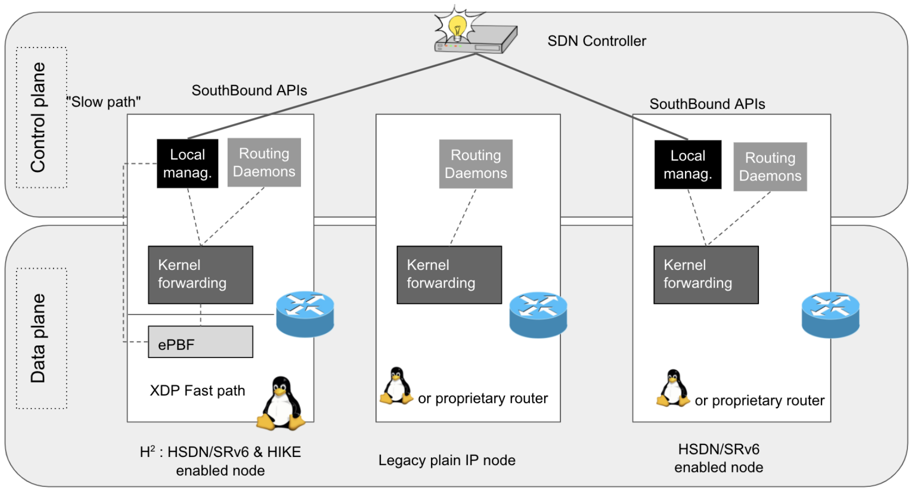


Figure 2: High level overview of the data plane/control plane architecture

3.3. The extended Berkeley Packet Filter (eBPF)

Proposed in the early '90s, the Berkeley Packet Filter (BPF) [9] is designed as a solution for performing packet filtering directly in the kernel. BPF comes with its own set of RISC-based instructions used for writing packet filter applications, and it provides a simple Virtual Machine (VM), which allows BPF filter programs to be executed in the data path of the networking stack. Starting from the release 3.18 of the Linux kernel, the Extended BPF (eBPF) [18] represents an enhancement over BPF, which adds more resources such as new registers and enhanced load/store instructions. It improves both processing and memory models. eBPF programs can be written using assembly instructions later converted in bytecode or in restricted C, and compiled using the LLVM Clang compiler. The bytecode can be loaded into the system through the `bpf()` syscall that forces the program to pass a set of sanity/safety-checks in order to verify whether the program can be harmful for the system. Only safe eBPF programs can be loaded into the system, the ones considered unsafe are rejected. To be considered safe, a program must meet a number of constraints such as limited number of instructions, limited use of backward jumps, limited use of the stack and etc. These limitations [19] can impact on the ability to create powerful network programs.

eBPF programs are designed to be stateless so that every run is independent from the others. The eBPF infrastructure provides specific data structures, called *maps*, that can be accessed by the eBPF programs and by the userspace when they need to share some information.

eBPF programs are triggered by some internal and/or external events which span from the execution of a specific syscall up to the incoming of a network packet. Therefore, eBPF programs are hooked to different type of events and each one comes with a specific execution context. Depending on the context, there are programs that can legitimately perform socket filtering operations while other can perform only traffic classification at the TC layer and so on.

3.4. the eXpress Data Path

The eXpress Data Path (XDP) proposed in [10] is an eBPF based high performance packet processing component merged in the Linux kernel since version 4.8. It preserves security thanks to a limited execution environment in the form of a virtual machine running eBPF code.

In the regular Linux kernel packet processing, the allocation of the so called socket buffer (`sk_buff`) that stores the packet data represents a performance hit. Therefore, XDP introduces an early hook in the RX path of the kernel, placed in the NIC driver, before any memory allocation takes place. This XDP operating mode is the so called "Native XDP" and creates an eBPF based high performance data path named eXpress Data Path (XDP). Every incoming packet is intercepted *before* entering the Linux networking stack and, importantly, before it allocates its data structures, foremost the `sk_buff`. This accounts for most performance benefits as widely demonstrated in literature (e.g., [20] [21]).

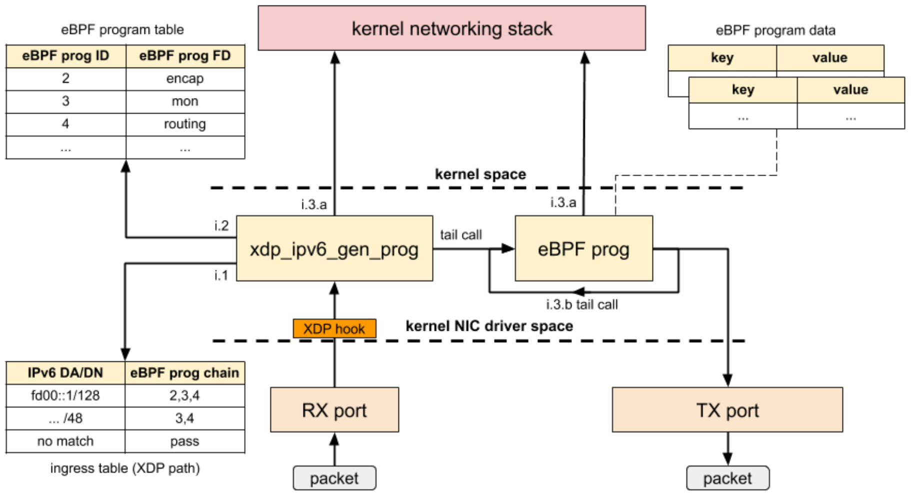
If the NIC driver does not support the “Native XDP” mode, the XDP program is loaded into the kernel as part of the normal network path (Generic XDP). However, the Generic XDP mode cannot achieve the same performance of Native XDP.

4. The HIKE Data Plane : HybrId Kernel/EBPF

The goal of HIKE data plane is to combine the conventional Linux networking kernel stack and eBPF/XDP in a flexible and modular way, taking the best from both worlds. An eBPF program allows to process packets to a very high data rate but it comes with some limitations that can prevent the parsing of nested packet headers, the arbitrary processing of packet payloads, the impossibility to use locks to handle concurrency and to avoid races, etc. Conversely, the “normal” Linux networking kernel stack is slower but it offers full control on the way in which packets have to be processed.

4.1. Overview

Figure 2 shows an architectural overview of the different components of the proposed architecture. Each node is a Linux device where the packet processing functionality is logically divided between a kernel path and fast path. The fast path includes the eBPF programs, capable of offering an efficient and high-performance packet processing by exploiting the XDP hook. The kernel path uses conventional routing and forwarding capabilities implemented by the Linux networking stack whose routing tables can be configured by hand or by using any routing daemon. Southbound APIs connect a Local Manager module with the SDN controller. The Local Manager is responsible for implementing the custom logic, configuring programs and behaviours both in the fast path and in the kernel path. Through the controller, the network administrator can thus *dynamically* enforce the decision on what kind of packets are going to be processed on a target set of nodes in the fast or kernel path. Figure 2 also highlights the difference between the proposed solution (H^2) and i) a legacy plain IP node (either Linux based or proprietary router) which is made up only by the conventional kernel routing/forwarding mechanism and the related routing daemons; and ii) an hybrid HSDN/SRv6 solution that does not leverage on eBPF/XDP.



eBPF program table

eBPF prog ID	eBPF prog FD
2	encap
3	mon
4	routing
...	...

eBPF program data

key	value
...	...

ingress table (XDP path)

IPv6 DA/DN	eBPF prog chain
fd00::1/128	2,3,4
... /48	3,4
no match	pass

Figure 3: HIKE eBPF/XDP Architecture

4.2. Full programmable node

In the Linux networking, conventional systems such as Netfilter/Xtables/IPtables offer a relatively high degree of modularity, allowing different filtering and processing jobs to be easily combined on the same packet.

In general, eBPF networking does not support such flexibility out-of-the-box. Indeed, eBPF includes several limitations imposed for performance and safety reasons: there is a maximum length of instructions allowed in a program (4096), the set of the kernel functions we can call inside an eBPF program (eBPF helper functions) are severely limited, and most importantly, eBPF is a non-Turing complete language (e.g., it is not possible to perform unbounded loops). All these limitations make it difficult to develop complex network functions as eBPF programs.

One way to improve the flexibility of eBPF is to develop basic functionalities and then combine them flexibly. However, the number of functions that can be concatenated is limited (i.e. we can chain up to 32 programs) and in addition this solution requires manual handling of program and high technical skills. For this reason so far eBPF programs have been often used mainly as fast but standalone/monolithic packet processors.

HIKE proposes a way to enhance eBPF/XDP, to support modularity and foster efficient function chaining, to create processing and filtering pipelines easily at high speed, and to dynamically append and remove processing functions to a node. The core of such architecture is in an efficient lookup (described in detail in section 4.3) that, starting from some features of the packet, calls a given eBPF program that in turn may call other eBPF programs through cascading *tail calls*.

In the considered Performance Monitoring scenario, we used IPv6 destination addresses and networks of IPv6 packets to perform such a match. However, in general this is applicable to an arbitrary set of matching rules and packet fields.

4.3. HIKE eBPF/XDP architecture

Figure 3 shows the eBPF/XDP architecture for HIKE. As we can see, packets enters from the RX port and are passed to the XDP hook. Once processed, packets exit from HIKE either through the TX port or because they are passed to the `kernel networking stack`. To better highlight which components are implemented in the kernel space and which ones are implemented in XDP, we draw a dividing line in figure 3: the components below the line stand in the kernel NIC driver space and benefit of the fast eBFP/XDP processing.

The packets are handled by the `xdp_ipv6_gen_prog` eBPF program [1] that performs an initial lookup (i.1) in the "ingress table" and, according to some packet features (e.g. the IPv6 destination address network in our case), gets the eBPF program chain which is composed of all the eBPF programs that must be used for processing the incoming packet. Each eBPF program in the chain is represented by an ID (numerical identifier) and the packet is processed by the eBPF programs accordingly to the order as they appear in the chain. For instance, a chain "2,3,4" imposes that the eBPF program "2" is executed on a packet before the "3" and the eBPF program "3" before the "4". Note that the execution priority of eBPF programs in a chain is based only on the order (from left to right) of the IDs and not on their value. If there is no associated eBPF program chain to a specific IPv6 destination address, the framework

[1] in further detail, packets are firstly scanned by a dedicated eBPF program which differentiates IPv4 from IPv6 packets, passing only the latter to the `xdp_ipv6_gen_prog` dispatcher program.

can provide a default eBPF program chain (if defined) or the packet is passed directly to the "normal" Linux networking stack.

When an eBPF program has to be executed, the HIKE needs to find out its descriptor. To cope with that, the framework specifies a dedicated "program table" (i.2) which contains the binding between every program ID and the relative file descriptor (FD). The FD is necessary for passing the control flow from the *caller* eBPF program to the *callee* eBPF program through a tail call. Therefore, the `xdp_ipv6_gen_prog` could be seen as a "trampoline" which parse the IPv6 packet, retrieves the eBPF program chain and transfers the control flow to the first to eBPF program in the chain.

An HIKE eBPF program can decide, according to its own internal logic, to pass the control to the kernel networking stack (i.3.a) or to tail call (i.3.b) the next eBPF program specified in the chain. The HIKE provides an *helper function* with the purpose of advancing the chain and executing (through a tail_call) the next eBPF program. However, this program chaining is limited by the maximum possible depth of the tail calls allowed in eBPF, which is currently set to 32.

HIKE programs are stateless but they can store and retrieve information in dedicated eBPF maps (as depicted in Figure 3). They could also rely on some already defined context variables which can be conveniently used to speed up implementation and processing. These variables (e.g., the offset at which the IPv6 header starts) allow clock cycles to be saved, for instance by avoiding parsing the packet twice. Passing such context is far from trivial in the eBPF world, since every program is a function with the same pre-defined prototype and cannot be modified. As a workaround we use a dedicated eBPF map as a per-cpu scratch area. The map is shared by all the different eBPF programs called in the chain.

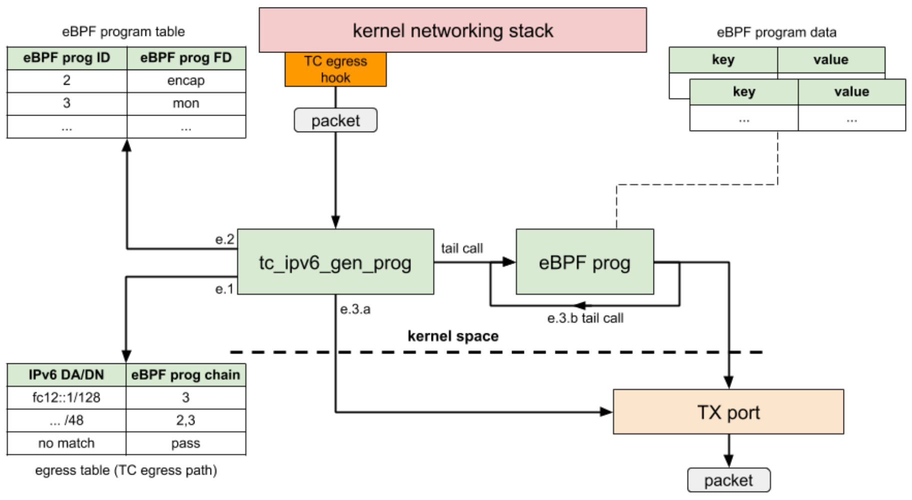
In the HIKE framework, thanks to the use of chains we can reuse and combine eBPF programs depending on the type of traffic that has to be processed. HIKE provides a number of prefabricated eBPF programs such as routing, SRv6 encap and monitoring.

The routing in HIKE is available in 3 different flavors and each of them is a dedicated eBPF/XDP program. The first routing program makes no use of the routing tables and helper functions of the Linux kernel. The performance of this routing program is remarkable because all the processing is carried out in the fast path without having to use the traditional Linux networking stack. The second routing program is based on a hybrid eBPF/XDP + kernel approach because it leverages the kernel routing tables using some helper functions. The third routing program is a totally kernel based approach where the packet is delivered to the Linux kernel networking stack which takes care of the routing and the forwarding.

Let us consider for example the ingress node of a SRv6 network (Node A in figure 1) . It receives packets that come from outside the SRv6 domain and must encapsulate, route and eventually monitor them. When a packet arrives, this is intercepted by the XDP hook, in which the eBPF program `xdp_ipv6_gen_prog` is loaded. This reads the IPv6 destination address and, accessing the ingress table, defines the sequence of programs to run (eBPF program chain). If, for example, the destination address matches the $fd00 :: 1/128$ entry this will be handled by programs 2,3 and 4 (encap, mon, routing). The program `xdp_ipv6_gen_prog` using a HIKE helper function starts the tail calls sequence by running the ”encap” program that encapsulates the IPv6 packet into another IPv6 packet with the Segment Routing Header (SRH). After the encap, the program that performs the monitoring tasks is invoked (e.g. counting and marking the packet) and finally the program that operates the routing determines the next hop and the associated output interface.

4.4. HIKE eBPF/TC egress hook

When the HIKE fast path cannot process a packet, the entire processing is on the behalf of the Linux networking stack. The Linux kernel can perform a plethora of different packet processing operations leveraging the power of Netfilter/Xtables/IPtables or custom kernel modules and TC. Custom operation can be executed by Kernel modules that have unconditional access to most of the internals of the system. However if there is a bug in the module, this can compromise the overall system security and stability. Thus the Linux kernel introduces the TC egress hook where eBPF programs can be attached. Aside with a slight performance advantage (far less than XDP), there are other benefits from developing the processing on eBPF such as the security and the avoidance of setting up a new dedicated kernel module. In this way it is possi-



eBPF program table

eBPF prog ID	eBPF prog FD
2	encap
3	mon
...	...

eBPF program data

key	value
...	...

IPv6 DA/DN	eBPF prog chain
fc12::1/128	3
... /48	2,3
no match	pass

egress table (TC egress path)

Figure 4: HIKE eBPF/TC Architecture

ble to perform some additional processing on the packets that are generated locally or forwarded to another hop. Therefore, in the HIKE architecture we introduced another programmable framework for processing packets leveraging the TC egress hook for attaching eBPF program. The HIKE TC framework reported in 4 is very similar to the eBPF/XDP framework discussed so far. Indeed, packets coming from the TC egress hook are processed by the `tc_ipv6_gen_prog` which retrieves (e.1) the eBPF program chains associated with the destination address of the IPv6 packet. If there is no eBPF chain, a default one (if present) could be used for finalizing the processing. Otherwise, the packet is forwarded directly to the tx port (e.3.a).

Due to the way in which eBPF handles programs in the different hooks, XDP programs used in the HIKE eBPF/XDP framework cannot be leveraged in the HIKE eBPF/TC one and vice versa. This implies that, for instance, the Encap eBPF/XDP program and the Encap eBPF/TC program are two distinct programs but they share some internal libraries and helper functions made available by the HIKE framework for avoiding code duplication as much as possible.

4.5. Monitoring System Implementation

To assess the performance of the HIKE architecture on a real scenario we implemented the loss monitoring framework presented in [22]. These were originally based on Linux kernel modules for carrying out all the necessary operations for parsing packets, updating flow counters and coloring.

We redesigned the programs so that they can be used stand-alone in the HIKE framework or chained together with other eBPF programs leveraging the flexibility of the program chains. The eBPF programs can be used for monitoring ingress and/or egress flows in a node that matches some given SR policies, also taking into account color marking.

All the hashtable data structures provided by the eBPF infrastructure support fixed length keys. Thus, to maximize the lookup performance and minimize memory waste, we created N maps/hashtables and in each map we store all the flows with the same SID list length. The value of N can be decided at compile time, and in our implementation the default max SID list length is equal to $N = 16$ SIDs. Therefore, we have come up with a total of 16 maps/hashtables.

The HIKE implementation we used for performance evaluation is available at [23]. An implementation of the full performance monitoring demonstrator is available as part of the ROSE ecosystem [24].

5. HIKE Performance Results

In this section we presents the performance of an HIKE implementation. The purpose of the performed experiments is to assess the impact of HIKE solutions in fast-path. In fact, the requirements of modularity and flexibility (such as the possibility to add programs on the fly or to reconfigure programs for a stream, etc.) require additional calls to programs and memory accesses with respect to the case of a static and optimized program. The devised test-bed aims to evaluate the impact of these operations which are built on top con conventional router operations.

Table 1: SUT Hardware characteristics

Type	Characteristics
CPU	2x Intel E5-2630v3 (8 Core 16 Thread) at 2.40 GHz
RAM	128 GB of ECC RAM
Disks	2x 1.2 TB HDD SAS 6Gbps 10K rpm 1x 480 GB SSD SAS 6Gbps
NICs	Intel X520 10Gb SFP+ Dual Port Intel I350 1Gb Dual Port

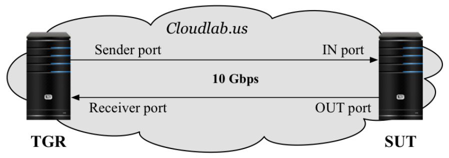


Figure 5: Testbed architecture

5.1. Processing load evaluation

We set up a testbed according to RFC 2544 [25], which provides the guidelines for benchmarking networking devices. Figure 5 depicts the used testbed architecture that comprises two nodes denoted as *Traffic Generator and Receiver (TGR)* and *System Under Test (SUT)* respectively. In our experiments, the packets are generated by the TGR on the Sender port, enter the SUT from the IN port, exit the SUT from the OUT port and then they are received back by the TGR on the Receiver port. Thus, the *TGR* can evaluate all different kinds of statistics on the transmitted traffic including packet loss, delay, etc. The testbed is deployed on the CloudLab facilities [26], a flexible infrastructure dedicated to scientific research on the future of Cloud Computing. Both the TGR and the SUT are bare metal servers whose hardware characteristics are shown in the table 1.

The SUT node runs a vanilla version of Linux kernel 5.6 and is configured as a specific node on this the SRv6 network. In the TGR node we exploit TRex [27] that is an open source traffic generator powered by DPDK [28]. We used SRPerf [29], a performance evaluation framework for software implementations, which automatically controls the TRex generator in order to evaluate the maximum throughput that can be processed by the SUT. The maximum throughput is defined as the maximum packet rate measured in Packet Per Seconds (PPS) for which the packet drop ratio is smaller then or equal to 0.5%. This is also referred to as Partial Drop Rate (PDR) at a 0.5% drop ratio (in short PDR@0.5%). Further details on PDR and insights about nodes configurations for the correct execution of the experiments can be found in [29].

5.2. HIKE tail call performance

We first configured the SUT with HIKE architecture analysing the performance and varying the number of the programs called with the tail calls. We loaded a static program chain: each program reads from the table the next program to call, and then executes it. The last program in the chain executes the packet forwarding by executing a redirect operation (i.e. without really executing the routing operation). In this way it is possible to evaluate the maximum throughput that the machine can support, at the net of the other operations that need be added to manage packets which necessarily imply a decrease in the maximum sustainable throughput by the node. In figure 6 we report the average maximum throughput vs the number of functions in the HIKE eBPF program chain. We executed 32 runs for each configuration, and we reported the standard deviation in the figure. The maximum throughput that the SUT can handle when a simple redirect is executed attains about 10.6 MPPS. When we add a tail of 8 programs to handle the packet, the maximum average throughput drops to 9.7 MPPS.

5.3. SRv6 Performance Monitoring

We then assess the HIKE performance in terms of throughput of some SRv6 Performance Monitoring configurations. To this aim, we implemented optimized eBPF programs to execute i) the encapsulation operation (ENC); ii) the packet counting and marking operations (MON); and iii) the routing/forwarding task (FW). We configured the SUT node to perform a combination of such functionality, and specifically: i) only

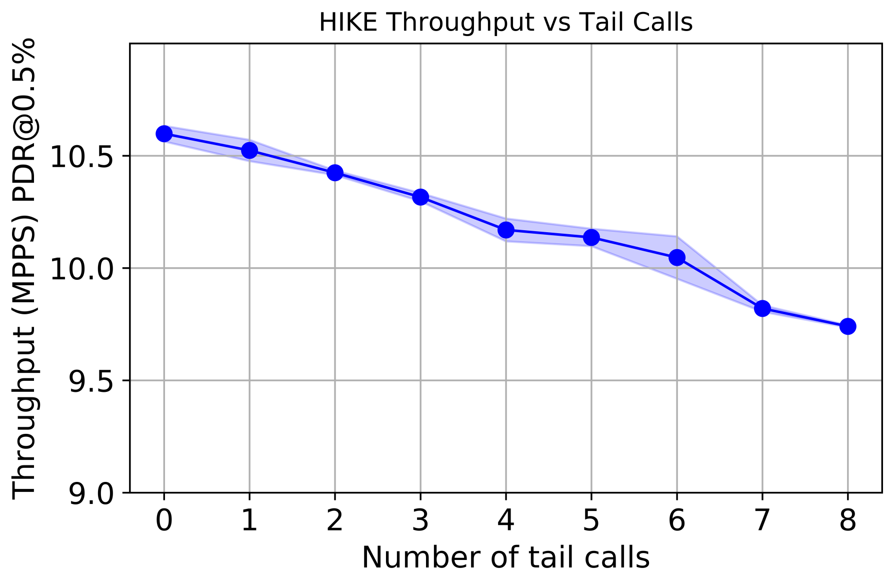


Figure 6: eBPF chain performance

the forwarding/routing task (1 tail call); ii) the monitoring and forwarding/routing tasks (2 tail calls); iii) the encapsulation and forwarding/routing tasks (2 tail call); iv) the encapsulation, monitoring and forwarding/routing tasks (3 tail call). In figure 7 we report the average SUT throughput for the four combinations of tail calls, varying the number of monitored flows. The figure also shows the throughput of the "Kernel Path" solution that executes the encapsulation and routing in the kernel stack. Moreover, we provided an upper bound implementing a pass through program, i.e. a program that forwards the packet to the output interface without modifying it.

As can be noted the measured throughput is independent from the number of monitored flows in all cases since all the solutions implement the hash-based matching described in [22]. The "Kernel Path" reaches a throughput of about 1 MPPS, and the "Upper bound" reaches about 5.1 MPPS. The configuration with only the forwarding/routing achieves the highest throughput which is around 5.0 MPPS, the configuration with monitoring and forwarding/routing 3.5 MPPS, the configuration with encapsulation and forwarding/routing 3.1 MPPS and the configuration with all the three

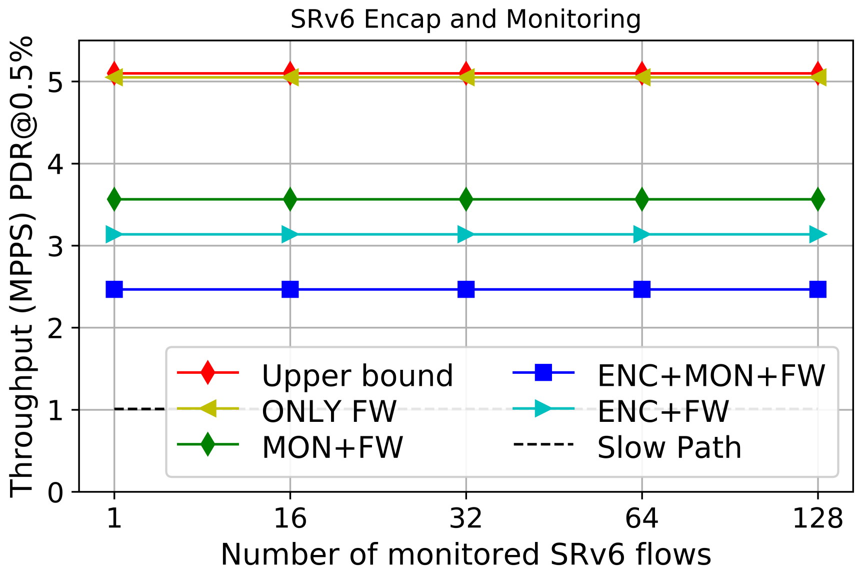


Figure 7: HIKE fast path vs kernel forwarding performance

operations combined 2.5 MPPS. The difference between the different solutions lies in the number of accesses to the eBPF maps (tables) they perform.

By comparing the maximum throughput achieved with the forwarding/routing programs, we notice a degradation from a baseline of $\sim$ 10 MPPS attained by the basic HIKE architecture reported in Figure 6, to $\sim$ 5 MPPS. This is accountable to the execution of the networking operations on the packets. However, as expected, the performance, compared to networking operated in the kernel, is about 5 times higher.

In our implementation there is a fixed cost of processing the packet plus a variable cost that depends on the packet length and on the length of the SID list. To assess the impact of this variable part, we measured the maximum throughput varying the number of SID in the list and considering two configurations: the ingress node that executes the encapsulation, monitoring and forwarding/routing tasks and the waypoint node that executes the monitoring and forwarding/routing tasks. We report the results in figure 8. As we can see, both node configurations experience a slight degradation. The ingress node achieves about 2.4 MPPS with one SID and about 2.1 with 4 SIDs.

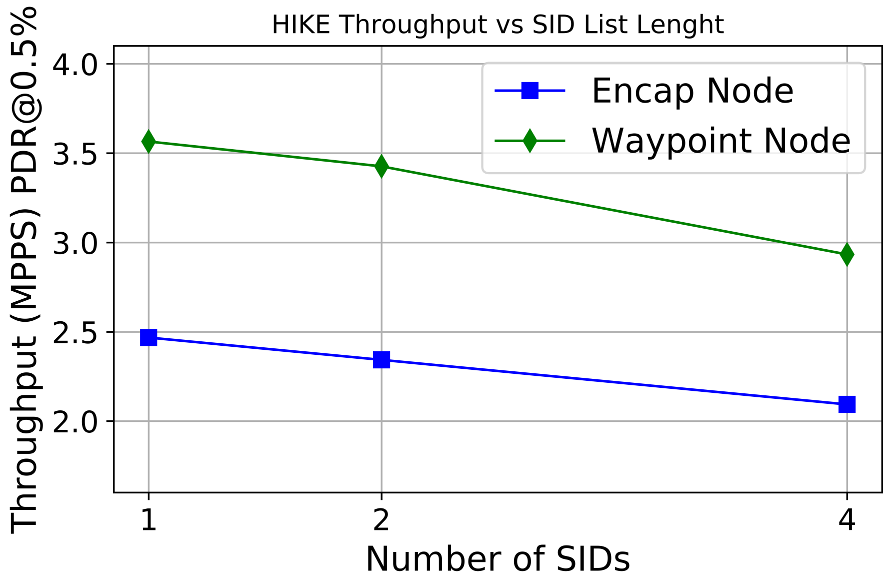


Figure 8: HIKE fast path performance with respect to the number of SID in the List

The waypoint node can handle about 3.5 MPPS with one SID and about 2.9 with 4 SIDs. As we can see by such analysis, we experiment i) a similar amount of degradation in the ingress and the waypoint configurations; and ii) a level of degradation that scales proportionally with the number of the SIDs, as a result of the (linearly increasing) amount of instructions needed to copy each SID in the packet.

6. Related Work

In this section we first review performance monitoring solutions proposed for HSDN networks. Then we present the related works on systems that uses eBPF for HSDN networks.

6.1. Performance Monitoring in HSDN networks

Performance monitoring of traffic flows in HSDN networks using traditional techniques poses several problems as highlighted in [30], because it is complicated to integrate information from traditional nodes with the information collected in the softwarized network. For example, in [31] the authors show how the delay in the acquisition of information from the traditional network causes a problem of convergence of the overall information seen by the controller, which can have a severe impact on traffic engendering decisions. In order to reduce this problem, techniques such as compressive sensing can allow the identification the main links of the network and can focus their monitoring on them, thus reducing estimation problems.

Solutions that are coming to prominence foresee the direct inclusion of monitoring and reporting functionality in the data plane. Some special packages are then used to manage and collect the measured data. These include techniques such as Inband Network Telemetry (INT) [32] and In situ OAM (IOAM) [33], but also more complex solutions such as Alternate Marking [16], which consists of coloring (marking) the packets of a flow with at least two different colors that alternate over time. This technique allows in-flight packets to be handles, thus rendering it difficult to obtain an accurate evaluation of the number of lost packets, as discussed in RFC 8321 [34]. To support the interoperability of different PM solutions, IETF has standardized some protocols for the packet loss and delay measurements and for collecting the measured data. Among them, RFC 4656 "The One-Way Active Measurement Protocol (OWAMP)" [35] provides capabilities for the measurement of one-way performance metrics in IP networks such as one-way packet delay, one-way packet loss and RFC 5357 "Two-Way Active Measurement Protocol (TWAMP)" [36] introduces the capabilities for the measurements of two-way (i.e. round-trip) metrics.

6.2. eBPF based solutions

In [20] authors propose eVNF, an hybrid virtual network functions with Linux express data path. In their architecture they use XDP for simple but critical tasks coarse packet filtering, leaving to the "slow path" (e.g., in user-space) the space for complex operations. They also tested their system building a firewall, a deep packet inspection module and a load balance. They did not consider the application of their system to performance monitoring. In [37], authors use the proposed node architecture to implement a firewall, a deep packet inspector and a load balancer, proving the flexibility of hybrid eBPF/kernel networking.

Also Abdelsalam et al. [38] deal with Linux firewall but using SRv6 and propose a SEgment Routing Aware (SERA) firewall which extends the iptables, allowing operations also on SR encapsulated packets. Differently from this approach they do not leverage on the XDP for performance improvement.

In [39] an interesting implementation is presented that uses eBPF to operate the PM in the Linux system. The authors show how the tables shared between eBPF and the kernel can be used to store information that is then read outside of eBPF. In our implementation we use a similar technique to control eBPF programs. However, the collection of measured data, in our case, is done using the TWAMP protocol.

In [40] it is shown how eBPF can be used to implement SRv6 instructions that are activated through lightweight tunnels in the "normal" Linux Kernel. eBPF in that case only provides the flexibility to insert the code directly into the kernel space, but the approach misses the node management architecture and the pure fast path eBPF. In [41] the same authors show how eBPF and SRv6 can be used to implement specific SDN features such as failure detection and fast reroute. This is a further demonstration of how network programming SRv6 can be successfully used to improve network flexibility.

7. Conclusions

In this work we considered a Linux software router in an Hybrid SDN/SRv6 architecture exploiting the eBPF (extended Berkeley Packet Filter). We propose a data plane architecture called HIKE (HybrId Kernel/EBPF forwarding) which integrates the packet forwarding and processing based on "normal " Linux kernel with the ones based on the eBPF platform, taking the best of both worlds.

The architecture provides an organisation of eBPF programs in the SDN context, overcoming the many technical limitations given by using the XDP fast path. We implemented the architecture in a proof-of-concept, and use SRv6 traffic monitoring as a test case. Numerical results show the performance improvement in terms of throughput with respect to using the Linux kernel networking stack. On commodity hardware we obtained 5.1 MPPS, which is around 5 times faster than the conventional solution.

The proposed architecture fosters a modular approach for combining different eBPF programs in to perform complex and customizable SDN packet processing.

Acknowledgments

This work has received funding from the Cisco University Research Program and from the EU H2020 5G-EVE project.

References

[1] C. Filsfils, et al., The Segment Routing Architecture, Global Communications Conference (GLOBECOM), 2015 IEEE (2015) 1–6.

[2] S. Previdi et al., Segment Routing Architecture, IETF RFC 8402 (Jul. 2018).
URL `https://tools.ietf.org/html/rfc8402/`

[3] SR is the de-facto SDN Network Architecture.
URL `https://www.segment-routing.net/conferences/2016-sr-defacto-sdn-network-architecture/`

[4] A. Bashandy et al., Segment Routing with the MPLS data plane, RFC 8660 (Dec. 2019). `doi:10.17487/RFC8660`.
URL `https://tools.ietf.org/html/rfc8660`

[5] C. Filsfils, D. Dukes (eds.) et al., IPv6 Segment Routing Header (SRH), RFC 8754 (Mar. 2020). `doi:10.17487/RFC8754`.
URL `https://tools.ietf.org/html/rfc8754`

[6] S. Matsushima et al., SRv6 Implementation and Deployment Status, Internet-Draft draft-matsushima-spring-srv6-deployment-status, Internet Engineering Task Force, work in Progress (Apr. 2020).
URL `https://tools.ietf.org/html/draft-matsushima-spring-srv6-deployment-status`

[7] SD-WAN.
URL `https://en.wikipedia.org/wiki/SD-WAN`

[8] D. Dukes et al., SR for SDWAN - VPN with Underlay SLA, Internet-Draft draft-dukes-spring-sr-for-sdwan, Internet Engineering Task Force, work in Progress (Jun. 2019).
URL `https://tools.ietf.org/html/draft-dukes-spring-sr-for-sdwan`

[9] S. McCanne, V. Jacobson, The BSD Packet Filter: A New Architecture for User-level Packet Capture, in: USENIX winter, Vol. 46, 1993, pp. 259–270.

[10] T. Høiland-Jørgensen, J. D. Brouer, D. Borkmann, J. Fastabend, T. Herbert, D. Ahern, D. Miller, The express data path: Fast programmable packet processing in the operating system kernel, in: Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies, 2018, pp. 54–66.

[11] C. Filsfils, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, Z. Li, SRv6 Network Programming, Internet-Draft draft-ietf-spring-srv6-network-programming-16, Internet Engineering Task Force, work in Progress (Jun. 2020).
URL `https://datatracker.ietf.org/doc/html/draft-ietf-spring-srv6-network-programming`

[12] P. L. Ventre, M. M. Tajiki, S. Salsano, C. Filsfils, SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks, IEEE Transactions on Network and Service Management 15 (4) (2018) 1378–1392.

[13] P. Loreti et al., SRv6-PM: Performance Monitoring of SRv6 Networks with a Cloud-Native Architecture, Submitted paper available on Arxiv (2020).
URL `https://arxiv.org/pdf/2007.08633`

[14] R. Gandhi, C. Filsfils, D. Voyer, M. Chen, B. Janssens, Performance Measurement Using TWAMP Light for Segment Routing Networks, Internet-Draft draft-gandhi-spring-twamp-srpm-08, Internet Engineering Task Force, work in Progress (Mar. 2020).
URL `https://datatracker.ietf.org/doc/html/draft-gandhi-spring-twamp-srpm-08`

[15] G. Fioccola, T. Zhou, M. Cociglio, F. Qin, IPv6 Application of the Alternate Marking Method, Internet-Draft draft-fz-6man-ipv6-alt-mark-09, Internet Engineering Task Force, work in Progress (May 2020).
URL `https://datatracker.ietf.org/doc/html/draft-fz-6man-ipv6-alt-mark-09`

[16] T. Mizrahi, G. Navon, G. Fioccola, M. Cociglio, M. Chen, G. Mirsky, Am-pm: Efficient network telemetry using alternate marking, IEEE Network 33 (4) (2019) 155–161.

[17] What is VPP ?, `https://wiki.fd.io/view/VPP`.

[18] Linux Socket Filtering aka Berkeley Packet Filter (BPF).
URL `https://www.kernel.org/doc/Documentation/networking/filter.txt`

[19] S. M. et al., Creating Complex Network Services with eBPF:Experience and Lessons Learned, in: IEEE International Conference on High Performance Switching and Routing (HPSR2018), IEEE, 2018, pp. 1–8.

[20] N. Van Tu, J. Yoo, J. W. Hong, evnf - hybrid virtual network functions with linux express data path, in: 2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS), 2019, pp. 1–6.

[21] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, G. Carle, Performance implications of packet filtering with linux ebpf, in: 2018 30th International Teletraffic Congress (ITC 30), Vol. 01, 2018, pp. 209–217.

[22] P. Loreti, A. Mayer, P. Lungaroni, S. Salsano, R. Gandhi, C. Filsfils, Implementation of accurate per-flow packet loss monitoring in segment routing over ipv6 networks, in: 2020 IEEE 21st International Conference on High Performance Switching and Routing (HPSR), IEEE, 2020, pp. 1–8.

[23] Hybrid Kernel/eBPF data plane for SRv6 based Hybrid SDN, available online at `https://github.com/netgroup/hike`.

[24] The ROSE ecosystem.
URL `https://netgroup.github.io/rose/`

[25] S. Bradner and J. McQuaid, Benchmarking Methodology for Network Interconnect Devices, RFC 2544, RFC Editor (March 1999).
URL `https://tools.ietf.org/html/rfc2544`

[26] Robert Ricci, Eric Eide, and the CloudLab Team, Introducing CloudLab: Scientific infrastructure for advancing cloud architectures and applications, ; login:: the magazine of USENIX & SAGE 39 (6) (2014) 36–38.

[27] TRex realistic traffic generator.
URL `https://trex-tgn.cisco.com/`

[28] Data Plane Development Kit (DPDK).
URL `https://www.dpdk.org/`

[29] A. Abdelsalam, et al., Performance of IPv6 Segment Routing in Linux Kernel, in: 1st Workshop on Segment Routing and Service Function Chaining (SR+SFC 2018) at CNSM 2018, Rome, Italy, 2018, pp. 414–419.

[30] A. A. Abushagur, T. S. Chin, R. Kaspin, N. Omar, A. T. Samsudin, Hybrid software-defined network monitoring, in: International Conference on Internet and Distributed Computing Systems, Springer, 2019, pp. 234–247.

[31] T. Y. Cheng, X. Jia, Compressive traffic monitoring in hybrid sdn, IEEE Journal on Selected Areas in Communications 36 (12) (2018) 2731–2743.

[32] J. Hyun, N. Van Tu, J. W.-K. Hong, Towards knowledge-defined networking using in-band network telemetry, in: NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium, IEEE, 2018, pp. 1–7.

[33] F. Brockners, S. Bhandari, T. Mizrahi, Data Fields for In-situ OAM, Internet-Draft draft-ietf-ippm-ioam-data-10, Internet Engineering Task Force, work in Progress (Jul. 2020).

URL https://datatracker.ietf.org/doc/html/draft-ietf-ippm-ioam-data-10

[34] G. Fioccola, Ed., Alternate-Marking Method for Passive and Hybrid Performance Monitoring , IETF RFC 8321 (Jan. 2018).
URL https://tools.ietf.org/html/rfc8321

[35] S. Shalunov, B. Teitelbaum, et. al, A One-way Active Measurement Protocol (OWAMP), IETF RFC 4656 (Sep. 2006).
URL https://tools.ietf.org/html/rfc4656

[36] K. Hedayat, R. Krzanowski, et al., A Two-Way Active Measurement Protocol (TWAMP), IETF RFC 5357 (Sep. 2006).
URL https://tools.ietf.org/html/rfc5357

[37] N. Van Tu, J.-H. Yoo, J. W.-K. Hong, Building hybrid virtual network functions with express data path, in: 2019 15th International Conference on Network and Service Management (CNSM), IEEE, 2019, pp. 1–9.

[38] A. Abdelsalam, et al., SERA: SEgment Routing Aware Firewall for Service Function Chaining scenarios, in: IFIP Networking 2018, IEEE, 2018, pp. 46–54.

[39] C. Cassagnes, L. Trestioreanu, C. Joly, R. State, The rise of ebpf for non-intrusive performance monitoring, in: NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, IEEE, 2020, pp. 1–7.

[40] M.Xhonneux, F.Duchene and O. Bonaventure , Leveraging ebpf for programmable network functions with ipv6 segment routing, in: Proceedings of the 14th International Conference on emerging Networking EXperiments and Technologies, ACM, 2018, pp. 67–72.

[41] M. Xhonneux and O. Bonaventure , Flexible failure detection and fast reroute using ebpf and srv6, in: 2018 14th International Conference on Network and Service Management (CNSM), IEEE, 2018, pp. 408–413.