UNIVERSITÁ DI ROMA TOR VERGATA
DOTTORATO DI RICERCA IN INGEGNERIA ELETTRONICA

NETWORK PROGRAMMABILITY IN SOFTWARE ROUTERS

DOCTORAL THESIS

Author
Andrea Mayer

Supervisor (s)
Prof. Stefano Salsano

The Coordinator of the PhD Program
Prof. Corrado Di Natale

June 2022
Academic Year 2020/2021

XXXIV Cycle

Questa tesi è dedicata a tutte le persone che mi amano ed hanno sempre creduto in me.

"A few first rate research papers are preferable to a large number that are poorly conceived or half-finished. The latter are no credit to their writers and a waste of time to their readers"
Claude Shannon

Ringraziamenti

In conclusione del mio percorso di dottorato ci terrei a ringraziare tutti coloro i quali mi hanno supportato/sopportato durante questo viaggio: dai colleghi che poi sono diventati amici, ai docenti e professionisti che hanno apprezzato le mie capacità e mi hanno spronato e dato spunti per migliorare. Un ringraziamento particolare va a al Prof. Stefano Salsano per l'aiuto, i confronti costruttivi, gli stimoli, gli spunti forniti, oltre che per la sua disponibilità mai venuta meno nel corso di questi anni. Infine, non meno importante, ringrazio la mia famiglia, Debora e gli amici che sono sempre stati al mio fianco.

Ancora grazie a tutti voi, di cuore.

Abstract

The joint deployment of Software Defined Networking (SDN) and Network Function Virtualization (NFV) architectures has enabled modern telecommunications networks and Cloud-Fog-Edge computing to accommodate a wide variety of user and service needs. The NFV architecture aims to virtualize network functions, traditionally implemented through dedicated and expensive hardware, so that they can be executed on general purpose hardware. At the same time, SDN brings programmability to networks by separating the control plane from the data plane. These two technologies are complementary and together define an effective solution to promote the software-based networking paradigm with significant benefits, i.e.: increased scalability, agility and innovation. However, the proper functioning of the entire network infrastructure requires a large amount of state to be stored within the equipment, hence including both edge and core routers. The massive presence of such state information in the network devices increases the complexity and reduces the efficiency of network (re)configuration operations (e.g. recalculation of routes, on-demand creation of VPNs, etc.).

Segment Routing for IPv6 (SRv6) is a hybrid SDN architecture based on source routing that avoids or dramatically reduces the need to add state information within core devices for deploying network services. In SRv6, a source node includes an ordered list of instructions (segments) represented by IPv6 addresses inside the Segment Routing Header (SRH). Segments can represent topological instructions used as waypoints to steer packets through a specific path within the network and to its final destination. At the same time SRv6, by implementing the Network Programming Model, considers segments as service-based instructions that can be exploited to indicate the operations to be performed by nodes once they receive packets.

Softwarization of network operator and data center infrastructures has paved the way for the use of SRv6, taking advantage of its network programming model. In this context, a key role within next-generation networks is covered by software routers. The majority of them are powered by Linux kernel which offers outstanding features in terms of compatibility of hardware, software, flexibility and programmability. The combination of SRv6 Network Programming Model and programmability of Linux

kernel-based network devices is a topic of great interest among researchers as well as network operators and cloud providers.

In this thesis work, I investigate and cover several aspects of network programming in software routers, mainly based on the Linux kernel, from various perspectives. Considering use cases of real interest to network operators and cloud providers, I investigate how a Linux-based router can be programmed by taking advantage of two macro concepts: i) the adoption of SRv6 Network Programming Model, providing the ability to encode, within specific headers, programs whose instructions are associated with functions run by routers, after receiving packets; ii) the possibility to extend network functions already offered by the Linux kernel and/or add new ones, either by modifying the source code or by using the packet processing frameworks available in the networking stack. These two concepts are related: having an easily programmable and efficient data plane becomes an enabling factor for the spread of network programmability, especially using SRv6. Efficiency plays a key role in the context of a software router, which is often totally virtualized on general purpose hardware or is implemented using hardware acceleration for only a few specific functions (i.e., checksum, fragmentation) and delegates all other operations to a general purpose CPU. In a software router, it is not only the number of supported network functions and protocols that matters, but also the performance that it is able to attain in executing complex network functions. In this thesis, I dedicate considerable attention to performance analysis. This is intended to determine the efficiency, based on certain metrics, of the architectures and of the implementations proposed to cope with the investigated use cases.

The different aspects related to network programming studied in this thesis work are grouped by topic areas and reported hereafter.

Evaluating performance for SRv6 implementations

SRv6 is considered one of the enabling technologies for both network operators and data centers, and for this reason is crucial to investigate non-functional characteristics such as performance and scalability. The SRv6 data plane is supported in different software routers such as the Linux kernel, Vector Packet Processor (VPP) as well as in hardware devices.

In this thesis work, I present SRPerf, a performance evaluation framework for SRv6 implementations. SRPerf can be used for carrying out different benchmarking tests with the purpose of evaluating throughput and latency. The SRPerf architecture can be extended to support other kinds of benchmarking tests and methodologies, as well as new SRv6 behaviors to be tested. During my research activity, SRPerf was an invaluable tool that I used for validating the performance of all my proposed solutions.

Performance Monitoring for SRv6 networks

Flexibility and reliability of modern high speed networks can be improved with the support of effective tools and systems aiming to monitor the health and performance of the infrastructure. A fully fledged Performance Monitoring system should consider three different aspects, at least: i) a data plane framework for measuring metrics such as packet delay and loss; ii) a control plane capable of initiating and terminate the measure operations; iii) big-data tools for collecting, processing, storing and visualizing results.

In my thesis work, I propose a Performance Monitoring architecture for SRv6 networks (SRv6-PM for short) that deals with all the aspects discussed above. SRv6-PM perfectly fits in a cloud-native architecture since it makes use of big-data tools for processing, visualizing performance monitoring data, and relies on an SDN-based control of the routers to drive performance monitoring operations. To validate the SRv6-PM architecture, I focus on loss monitoring and consider a solution capable of tracking per-flow packet loss events operating in near-real time. I referred to this solution as the Per-Flow Packet Loss Monitoring system (PF-PLM). I provide different implementations for the PF-PLM, each one considering different design choices. These choices depend primarily on the availability of a packet processing framework (i.e. Netfilter/Xtables, extended Berkeley Packet Filter - eBPF) in Linux-based software routers and the ease of programming within the specific framework. The performance of the different implementations was measured by considering the overhead introduced by monitoring operations compared with the baseline solution without any active monitoring. The best results are obtained with eBPF, whose monitoring operations introduce about 5% overhead compared to the baseline. However, programming in eBPF requires more skills than using frameworks such as Netfilter and appropriate firewall rules.

Extending and improving the SRv6 Network Programming framework in Linux-based routers for supporting multi-site/tenant VPNs, Function Chaining for Legacy network functions, SID compression (Micro SID)

The implementation of the SRv6 network programming model available in Linux kernel-based routers is not complete, i.e. not all the functionality defined in the the IETF standards and in the stable Internet Drafts of IETF Working Groups have been implemented. In particular, the behaviors required to implement multi-tenant IPv4/IPV6 VPN solutions, for interconnecting users and applications between different sites with access to public/shared networks, are not implemented. This makes Linux-based routers not directly usable in these scenarios and thus forces network operators and cloud providers to fall back on proprietary and expensive hardware or software solutions that are not integrated into the kernel. However, limitations are more widespread. In the NFV domain, a Linux-based router does not provide support for network programs that, by implementing function chaining via SRv6, make use of SRv6-unaware (or legacy) functions. This makes legacy functions in the NFV domain, which are generally of great value to operators, totally unusable. Finally, no network function appears to be implemented in the Linux kernel to handle the compression of SIDs, through efficient representation of them. Having more compact IPv6 headers, in terms of overall length, helps interoperability between Linux-based software routers and disparate hardware devices, whose header processing is limited to a fixed number of bytes.

In this thesis, I propose a number of improvements and extensions, including architectural ones, to the Linux kernel with the aim of extending support for the SRv6 Network Programming Model in the above scenarios. My goal is to make routers based on the Linux kernel ready to support network programming in as many cases as possible, especially those of greatest interest to network operators. Through a series of patches, I provide support for SRv6 multi-site and multi-tenant IPv4/IPV6 VPNs,

revisiting the architecture of some core system components such as the Virtual Routing And Forwarding (VRF) of the Linux kernel. I devise and evaluate an SRv6 proxy architecture called SRNK (SR Proxy Native Kernel) to support the SFC of legacy network functions, providing several implementations (SRNKv1 and SRNKv2) seamlessly integrated into the Linux kernel. The overhead introduced by the SRv6 Proxy (SRNKv2) was measured and found to be marginal, barely a 3.5% of additional overhead compared to the reference (no proxy) case. Finally, I design and evaluate a new compression solution of SIDs (Micro SID) integrated into the kernel. Not only the compression can save tens of bytes in the SRH for each packet, but it also leads to a performance gain in throughput, a 2% increase in comparison to the case where no compression technique is applied.

Devising, designing and implementing high-performance programmable data plane architectures for Linux-based software routers in (Hybrid) SDN-based networks

SDN-based networks bring a set of important advantages in networks such as, reduction of operational costs, simplification of management operations, and most importantly, network programmability. The introduction of SRv6 with its hybrid SDN architecture enables full exploitation of IP standard routing and forwarding in both the control plane and the data plane. By leveraging the SRv6 Network Programming Model, network programmability takes a quantum leap as it becomes possible to encode programs through instructions (SIDs) embedded in packet headers. These instructions are matched by network functions, available in the data plane of the nodes, which are executed when SRv6 packets are received. Consequently, the greater is the number of functions provided for network programs, the more incisive and expressive is the SRv6 network programming model.

In this thesis work, I design and evaluate an efficient Linux-based software router that makes use of a programmable data plane architecture called HIKe-v0 (short for HybrId Kernel/eBPF forwarding) in an HSDN/SRv6 scenario. This architecture integrates the conventional Linux kernel packet forwarding with custom eBPF/XDP (extended Berkeley Packet Filter on eXpress Data Path) and eBPF/TC (eBPF on Linux Traffic Control) programs to speed up performance and extend capabilities of SRv6 software routers. The HIKe-v0 architecture provides an organization of eBPF programs in the SDN context, overcoming several technical limitations given by using eBPF in XDP and TC contexts. Specifically, HIKe-v0 facilitates an eBPF developer in defining lists of eBPF network programs (actually "program chains") which are applied automatically and sequentially on packets that match some user-defined criteria. However, HIKe-v0 still exposes the quirks and the complexities in writing eBPF programs to developers who should already be familiar with the eBPF world. In this regard, I devise and propose an evolution of HIKe-v0, named HIKe-v1. This time, HIKe-v1 stands for "Heal, Improve and desKill eBPF" and aims to enable novice developers to benefit from the performance and flexibility of eBPF packet processing, without dealing with eBPF programming. HIKe (referring to HIKe-v1, from now on) offers a Virtual Machine abstraction (HIKe VM) which facilitates the composition of eBPF programs using a programmatic approach. Using HIKe, a set of pre-fabricated eBPF programs can be composed in the so-called "HIKe Chains" using the standard C language. Finally, I mapped the HIKe abstraction into a high level language and

programming framework named eCLAT (eBPF Chains Language And Toolset), with the goal of further simplifying the work of a developer. In fact, a developer can write eCLAT scripts in a python-like language, composing HIKe eBPF programs, with no need of understanding the complex details of regular eBPF programming.

Contributions

My research activity has been mainly based on the theory and practice of network programmability, as well as the programming of software-based network equipment such as routers. In this thesis, I focused on many different activities related to network programming that could be categorized in: i) the design and implementation of new network functions to improve and extend the SRv6 Network Programming Model in Linux; ii) the research, design, implementation of programmable architectures for packet processing considering the different frameworks and technologies present within software routers, mostly based on the Linux kernel. However, I went even further. Where existing network packet processing frameworks were unable to offer the appropriate flexibility, functionality and performance, I conceived, designed, and implemented novel processing systems that allowed me to provide solutions to problems not yet fully solved.

Below, I report the major contributions of my research activities covered by this thesis.

Improving and extending the SRv6 Linux kernel implementation

The Linux kernel can be considered a full-blown software router with a plethora of services implemented within. Flexibility, great hardware support, and robust network configuration tools make the Linux kernel the ideal tool for powering software routers. The more network services and features are implemented in the Linux kernel networking stack, the more use cases network operators and administrators can address using a single "tool". With the rise of network softwarization, the role of Linux as the kernel for network operating systems used by software routers has become even more important, i.e. the SONiC operating system. In addition, the advent of HSDN/SRv6 architectures has made it possible to push network and device programming to unprecedented levels. Support for SRv6 in the Linux kernel, before my intervention, was partial and with structural problems that prevented the addition of new capabilities. My contribution on SRv6 in the Linux kernel has been considerable over the years, and I briefly summarize some of the major features I have introduced hereafter:

- SRv6 Behaviors did not accept optional attributes/parameters during configuration.

This was a big issue since it prevented the support for all those network functions (i.e. behaviors) that required extra attributes for being properly instantiated. For this reason, I re-designed the way in which the kernel handles SRv6 behavior attributes by extending the support also to the optional ones. It was not an easy work at all, since I had to guarantee that such a deep structural improvement would not break compatibility with previous kernel releases. Legacy support is always a pain, indeed;

- I extended the SRv6 subsystem to support optional callbacks (constructor/destructor) for customizing the creation/destruction of SRv6 behaviors. This capability was missing in the kernel and it prevented network functions to allocate/de-allocate required resources, i.e. memory, acquiring/releasing reference counters, etc;
- Optional attributes and custom constructor/destructor callbacks for SRv6 behaviors were the basis on which I devised and introduced SRv6 Counters. They are mainly used for obtaining statistics from SRv6 behaviors, i.e. number of packets processed successfully, number of packets dropped, errors, etc. However, on several occasions they have proved to be very useful tools for troubleshooting misconfigured SRv6 newtworks, since before them there was no way to guess whether a behavior was not triggered or was not properly set;
- I extended the Virtual Routing and Forwarding (VRF) of the Linux kernel to keep track of the relationships between the VRF devices and the routing tables associated with them (*VRF Strict mode*). There are conditions for which a one-to-one relationship must be satisfied between a VRF and the corresponding routing table. The lack of this property was an open issue for a long time. I finally took the chance to fix it, since it was a fundamental requirements for the upcoming kernel features that I would have introduced later;
- On top of the VRF Strict mode patch, I implemented the missing SRv6 End.DT4 behavior which is heavily used for creating multi-tenant IPv4 L3 VPNs solutions based on SRv6. The End.DT4 makes use of a VRF device to properly route traffic to the right tenant. Furthermore, I also patched the existing SRv6 End.DT6 behavior to support VRF mode as well. Listening to the needs of the SONiC community, I ended by providing the support for the End.DT46 behavior which provides the implementation of dual stack IP L3 VPNs. This behavior helps operators and administrator to simplify the configuration for L3 VPNs in scenarios where both IPv4 and IPv6 protocols should be considered;
- Finally, I decided to introduce in the Linux kernel the support for Micro SID extension to the SRv6 Network Programming Model. In this way, the Linux kernel can benefit from SRv6 compression header techniques with the desirable effect of improving the interoperability with those network devices that can not handle long SID Lists encoded in the Segment Routing Header (SRH);
- Finally, I decided to introduce in the Linux kernel the support for Micro SID extension to the SRv6 Network Programming Model. In this way, the Linux kernel can benefit from SRv6 compression header techniques with the desirable effect of improving the interoperability with those network devices that can not handle long SID Lists encoded in the Segment Routing Header (SRH);

It is worth noting that all the solutions I just presented above (except for the Micro SID which is under submission) have been appreciated by the community and they have

become part of the official Linux kernel code. Millions, if not billions of Linux-based devices nowadays can leverage the new network functionalities that I have introduced in the Linux kernel for accomplishing different tasks such as multi-tenant SRv6 VPNs and performance monitoring through counters.

Fixing bugs in the SRv6 subsystem of the Linux kernel

During the design and implementation of the new network features mentioned above, I faced several bugs. First, I had to understand their nature and then fix them later with several patches. The types of bugs I found were related to the way packet metadata was being accessed by different kernel functions. Those functions did not take into account that some pointers to memory areas, after certain types of operations on packets, could point to addresses that were no longer valid. The effects of these bugs are terrible: difficult to find, and they can cause the system to panic if they point to memory areas that are no longer allocated or invalid. Another class of bugs that I have frequently found is related to memory leaking.

Fixes to the problems identified have been accepted in the Linux kernel mainline;

Designing a performance evaluation framework for SRv6

SRv6 is considered one of the enabling technologies for both network operators and data centers, and for this reason is crucial to investigate non-functional characteristics such as performance and scalability. To this regard, I co-designed and co-implemented a performance evaluation framework for SRv6 implementations, named SRPerf. The design of such a framework was a very challenging task because packets are required to be forwarded at an extremely high rate using a limited CPU budget to process each of them. My contributions to the SRPerf framework can be summarized as follows: i) I participated in the design of the overall SRPerf architecture; ii) I provided the evaluation methodology and implemented the algorithms required to estimate performance metrics such as Non Drop Rate (NDR) and Partial Drop Rate (PDR); iii) I designed and implemented the low-level drivers for interfacing the SRPerf with the traffic generator used for testing the performance of the System Under Test (SUT).

Designing and implementing a solution to integrate legacy network function into the SRv6-based SFC architecture

One of the main features of SRv6 consisting in the native support of NFV/SFC scenarios. The SRv6 Network Programming Model provides the possibility to chain (virtual) functions in a network program embedded in the packet header. To this regard, I designed and implemented a *dynamic* proxy mechanism into the Linux kernel to support Service Function Chaining based on SRv6 for legacy VNFs (also called SRv6-unaware VNFs), a use case of great importance for service providers. Indeed, legacy VNFs have been in use since long time and network operators have spent a lot of effort to automate their deployment and operations. The SRv6 Proxy processes the SRv6 information on behalf of the Legacy VNFs: it removes the Segment Routing Header when packets are delivered to Legacy VNFs and restores it back when packets are sent from the VNFs. I came up with two versions of the SRv6 Proxy which are called SRNKv1 and SRNKv2. I identified a scalability issue in the first design SRNKv1, which had a linear degradation of the performance with the number of VNFs to be supported. This is due to the way the Linux Routing Policy subsystem is implemented and how it was

used for accessing the SRv6 information to be added/removed to/from packets. The final design SRNKv2 solved the problem, by extending the Policy Routing framework with the introduction of a new rule specific for SRv6 aimed to improve the access to SRv6 information to be restored in packets coming from VNFs.

To configure the SRv6 Proxy, I also patched the `iproute2` tool suite to implement the required commands. I released both the SRv6 Proxy and the `iproute2` as open source and, thus, they are freely available.

Designing and implementing solutions for supporting Performance Monitoring of SRv6 networks

Novel paradigms such as SDN, NFV and Network softwarization can increase flexibility, reliability of high speed networks when they are supported by effective tools and systems able to monitor the health and performance of the infrastructure. To this regard, I co-designed an architecture, named SRv6-PM, for supporting performance monitoring of SRv6 networks. The SRv6-PM architecture covers all the aspects that a full blown Performance Monitoring solution should have: i) a data plane framework for measuring metrics such as packet loss, delay; ii) control plane framework for driving performance monitoring operations; iii) big-data tools for collecting, processing, storing and visualizing performance monitoring data. To validate the SRv6-PM architecture, I designed and implemented an accurate Per-Flow Packet Loss Measurement (PF-PLM) framework based on the extension of the TWAMP protocol and the alternate marking techniques. The design of PF-PLM was a very challenging task because: i) the solution should be capable of tracking single packet loss events operating in near-real time (e.g. with a time granularity in the order of 10-20 seconds); ii) packets must be counted and marked at an extremely high rate using a limited CPU budget to process each of them; iii) monitoring activities must not introduce too much overhead on forwarding operations.

I designed and realized two PF-PLM solutions based on the Netfiter packet processing framework built into the Linux kernel. My first solution combined Netfilter with Iptables, but it did not scale as the number of traffic flows to be monitored increased. This is due to the way Iptables handles firewall rules (i.e. flows to be monitored). For this reason, I came up with a second implementation of PF-PLM leveraging the *IP set* framework, still based under the hood on Netfilter. *IP set* was not designed to handle SRv6 Policies and, thus, I extended it to support them. This time, the overhead introduced by the monitoring solution did not depend on the number of the monitored SRv6 flows. Even though the *IP set* implementation scales as the number of monitored flows increases, the overall overhead introduced by the monitoring was roughly about 15.0%, which is by no means negligible.

In the attempt to further reduce the overhead, I moved to a different design based on a new technology that exploits an in-kernel virtual machine (eBPF VM) able to run network programs, with very little overhead. With this approach, packet marking and counting were carried out by ad-hoc eBPF programs. In this case, I was able to achieve a significant performance increase over the *IP set* implementation, getting closer to the basic maximum throughput achieved for a flow that is not monitored. Indeed, the overall overhead introduced by the PF-PLM eBPF based solution was less than 5.0%.

Designing and implementing a high-performance programmable data plane architecture for Linux-based software routers in HSDN/SRv6 networks

Hybrid SDN architectures such as SRv6 can push network programmability to the limits when they can rely on flexible, customizable and easy-to-program software routers. For this purpose, I designed and implemented an efficient programmable data plane architecture to support HSDN/SRv6 networks which can be deployed on Linux-based software routers. This architecture, named HIKe-v0 (HybrId Kernel/eBPF forwarding, version 0), integrates the conventional Linux kernel packet forwarding with custom designed eBPF (extended Berkeley Packet Filter) programs, hooked into the eXpress Data Path (XDP) and Linux Traffic Control (TC) to speed up performance of SRv6 software routers. The HIKe-v0 architecture provides an organization of eBPF programs in the SDN context, overcoming the several technical limitations given by using the XDP and TC. HIKe-v0 facilitates an eBPF developer in defining lists of eBPF programs to be applied sequentially on packets that match user-defined criteria. Making an analogy between Iptables and HIKe, it can be said that the former offers the possibility of defining sequences of rules (grouped in chains) to filter traffic, while the latter provides the infrastructure to apply a sequence of eBPF programs on packets. In fact, such sequences of programs can be represented as a "chain", thus defining the concept of "program chain" absent in the eBPF core. Once I outlined the architecture of HIKe-v0, I progressed in creating an initial proof-of-concept where I took SRv6 Performance Monitoring (PM) as the use case. Then, I tested the HIKe-v0 PM implementation and the numerical results revealed a remarkable performance improvement in terms of throughput (5x more) compared to that achieved using only the "traditional" Linux kernel networking stack.

Devising, designing and implementing an innovative framework to make eBPF network programming reachable to everyone

With the rise of the Network Softwarization era, the extended Berkeley Packet Filter (eBPF) has become a hot technology for efficient packet processing on commodity hardware. There exist production-ready tools based on eBPF that offer unrivaled performance capabilities, such as the Cilium Framework. However, the development of custom eBPF solutions is a challenging activity that requires highly skilled human resources, and this fact hinders the full exploitation of eBPF potential. I identified the major obstacles and shortcomings faced by eBPF developers and I proposed the HIKe-v1 framework. HIKe-v1 is an evolution of HIKe-v0, it introduces many interesting and exciting new features that make eBPF network programming an enjoyable and productive experience even for developers who are not eBPF wizards. In this context, HIKe-v1 stands for *Heal, Improve and desKill eBPF* which I will call it simply HIKe from now on. The most important feature I introduced in HIKe was the possibility to compose/chain eBPF programs together following a custom logic and not necessarily a linear one (as in case of HIKe-v0). This custom logic is described using the C language, forming the HIKe Chain. A HIKe Chain appears to a programmer like a *block of code*, where he/she can make use of variables, branch conditions, loops and, most importantly, can invoke eBPF programs exploiting the well-known *function call* paradigm. Moreover, a HIKe Chain can leverage the same *function call* paradigm to invoke also other HIKe Chains. HIKe Chains are compiled into a bytecode run by

the Virtual Machine (HIKe VM) that I designed, implemented and provided within the HIKe framework. The HIKe VM can be embedded into eBPF programs, so that the eBPF VM can execute the HIKe VM that, in turns, interprets and executes the bytecode of HIKe Chains. The novelty of HIKe lies in the fact that HIKe Chains do NOT have to be verified by the eBPF verifier to be loaded and run, bringing significant advantages in terms of faster design and prototyping of complex network functions. In this way, pre-fabricated eBPF programs, written by eBPF experts for HIKe, can be combined (i.e. chained) with each other without any changes to their source codes and, thus, without having to re-verify them again. To my current knowledge, no other framework provides such features that allow end users to exploit fully customized eBPF solutions without writing eBPF code!

The eCLAT framework was developed with the aim of simplifying the management of the entire HIKe framework and, at the same time, providing a Python-like scripting language for coding HIKe Chains. eCLAT helps inexperienced eBPF programmers (Muggles) in the creation of complex network applications by composing/chaining eBPF programs already realized by eBPF experts (Wizard). Using eCLAT, a Muggle does not have to use C-language to implement HIKe Chains (i.e. to chain programs), does not have to know the internals of the HIKe framework or even the eBPF core. In other terms, a Muggle can implement high-performance and complex network services without having to know a single line of C code nor any details about eBPF internals. To test the performance of HIKe, I considered a real use case scenario related to a network under DDoS attack. A mitigation network service was implemented using *IP set* and HIKe. The HIKe based DDoS solution outperforms the *IP set* based one whose performance is roughly 47% worse than HIKe, i.e. *IP Set* wastes fare more resources such as CPU cycles, bandwidth, energy, etc.

Contents

CHAPTER 1

An Open Source ecosystem for SRv6 Network Programming Model

Network operators and cloud providers and are struggling to deal with the compelling requirements coming from evolving IP networks, just for citing a few: i) the integration of tens of billions of devices and services in the cloud, ii) the support for high capacity and low latency interconnections, iii) the support for advanced network services that can increase network security based on prevention and detection of unauthorized intrusions. Regardless of these challenging requirements, network operators also need to minimize the operating and capital spending associated with providing their services.

The network softwarization, with Software Defined Networking (SDN) [38], [57] and Network Function Virtualization (NFV) [121] approaches, has enabled networks to make a quantum leap forward in terms of cost reduction, flexibility and scalability. In this context, network operators start replacing physical network infrastructures with cloud-based systems which can be dynamically instantiated on demand based on the required level or service and performance needs. Highly specialized hardware appliances are replaced with software-based Virtual Network Functions (VNFs) that can be distributed and run on a virtualization infrastructure (Network Function Virtualization Infrastructure - NFVI) built upon commodity off-the-shelf (COTS) hardware.

Modern data center are fastly moving from specialized hardware towards industry-standard switching/routing silicon, COTS hardware, general purpose CPUs and Linux-based operating system (Linux-based OSs run on more than 99.0% of the Top 500 Supercomputers, [1]). This combination makes networking more scalable, more flexible, cheaper and more adaptable to the changing needs and requirements of the network and data center business.

As a fundamental complement to the network softwarization, I present the Segment Routing over IPv6 network architecture (SRv6) [53]. SRv6 offers great benefits to network operators, cloud operators and service providers aiming to: i) simplify the network by removing protocols and making network operations easier; ii) make the network more robust and resilient; iii) improve network performance by dynamically rerouting traffic; iv) chain together different services in order to implement complex services.

In this Chapter, I provide an overview on the Segment Routing for IPv6. In particular in section 1.1, I introduce the SR architecture by considering both the data plane as well the control plane. Along with that, I also describe the Network Programming Model for the Segment Routing IPv6-based solution and the Micro SID extension introduced for header compression purposes. I conclude this Chapter briefly describing in section 1.3 an open source SRv6 ecosystem which can be used for experimenting with SRv6 technology and for developing and evaluating new SRv6 features. Such ecosystem goes under the name of ROSE (Research on Open SRv6 Ecosystem) [137] project in which I am involved as an active author and contributor.

1.1 Segment Routing (SR)

Segment Routing (SR) [54, 148] is based on the *loose Source Routing* concept. With Segment Routing, a node can include an ordered list of instructions (known as *segments*) in the packet headers. The segments can represent: i) topological instructions used as way-points for steering the forwarding of packets through a specific path in the network and towards its final destination; ii) serviced based instructions which can be exploited for indicating the operations to be performed by the receiving nodes. List of instructions to be applied to a packet are carried in the so-called *Segment List* which is also referred as *SR Policy*.

The SR architecture can be deployed on two different data planes: i) SR over MPLS (SR-MPL) [4] or ii) SR over IPv6 (SRv6) [25]. The MPLS-based solution reuses the MPLS forwarding engine of IP routers (mostly backbone ones) based on MPLS labels. Hence, the semantic of MPLS labels is changed to represent the segments and the MPLS control plane needs to be updated accordingly. Further details on SR-MPLS are available in [173].

In this thesis, I consider the most recent implementation of SR based on IPv6. In this case, segments are IPv6 addresses and they are carried by a new Extension Header called Segment Routing Header (SRH) which extends the IPv6 protocol. Thanks to the way the IPv6 have been designed, routers that are not capable of processing the SRH still continue to work and forward the traffic by considering only the IPv6 destination address. In other terms, even cheap routers (and not only the expensive backbone ones) can forward SR IPv6-based traffic; on the contrary routers which are not design to handle the MPLS labels cannot process SR MPLS-based traffic properly, and they have to fallback on a default action or, even worse, drop the traffic.

However, regardless of the type of SR used, the most important advantage of the Segment Routing architecture resides in its high flexibility and scalability compared to traditional connection-oriented architectures for IP backbones (e.g. traditional MPLS).

1.2 Segment Routing over IPv6

In this Section, I introduce the concepts of the SRv6 control plane in Subsection 1.2.1, the SRv6 data plane in Subsection 1.2.2 and the SRv6 Network Programming Model (NPM) in Subsection 1.2.3. An in-depth survey on Segment Routing can be found in [173].

1.2.1 SRv6 control plane

The SR control plane, in the SR architecture, is in charge of configuring the SR data plane by accomplishing two main actions: i) configuring both the traffic classification and the *SR Policies* to be applied on packets at the SR source nodes; ii) to handle the association between Segment Identifiers (SIDs) and instructions to be executed in a given node.

Routing protocols such as OSPF, BGP are used to distribute both routing information for reachability purposes as well as basic SRv6 topological instructions (i.e: waypoints) in the same routing domain. Conversely, routing protocols have been extended for distributing specific SRv6 information such as the binding between SIDs and SRv6 Behaviors or even the node capabilities. Once the SR controller is aware of the network topology and the capabilities of the SRv6 nodes, it can take decisions and program SR source nodes and SR endpoints according to the SDN principles. In order to program the SRv6 data plane, the SR controller can rely on top of standardized southbound interfaces (i.e.: BGP, CONF).

1.2.2 SRv6 data plane

Data plane of the Segment Routing over IPv6 relies on a new type of IPv6 Extension Header called Segment Routing Header (SRH) which is defined in [25]. The SRH makes use of the *Segment List* field for containing the ordered list of IPv6 addresses used for steering the packet along a specific path in the network. Each IPv6 address contained in the Segment List is called *Segment Identifier* (SID). The SRH contains also the *Segment Left* field which specifies the number of remaining SIDs that still need to be traversed by the packet before being delivered to its final destination. The Segment Left is used as a pointer since its value indicates the *next active* segment. Instead, the active segment is indicated by the Destination Address (DA) of the packet. The complete format of the SRH is shown in Figure 1.1.

The SRH processing in the SRv6 domain is based on three different types of SRv6 nodes [25]:

- **SRv6 Source node**: can be a host originating a packet or an ingress node of an SRv6 domain. Either way, the Source node encapsulates the packet into an new outer IPv6 packet followed by the SRH containing one or more SR policies. The first SID in the Segment List associated with a SR policy is encoded as the packet destination address. At this point, the packet is forwarded towards its destination address;
- **SRv6 Transit Node**: forwards SRv6 traffic to the next hop in the path towards the destination, since the destination address of the packet is not configured as (local) SID. Transit nodes can be SR-capable or SR-incapable and they are not required to inspect the SRH of received SRv6 packets;

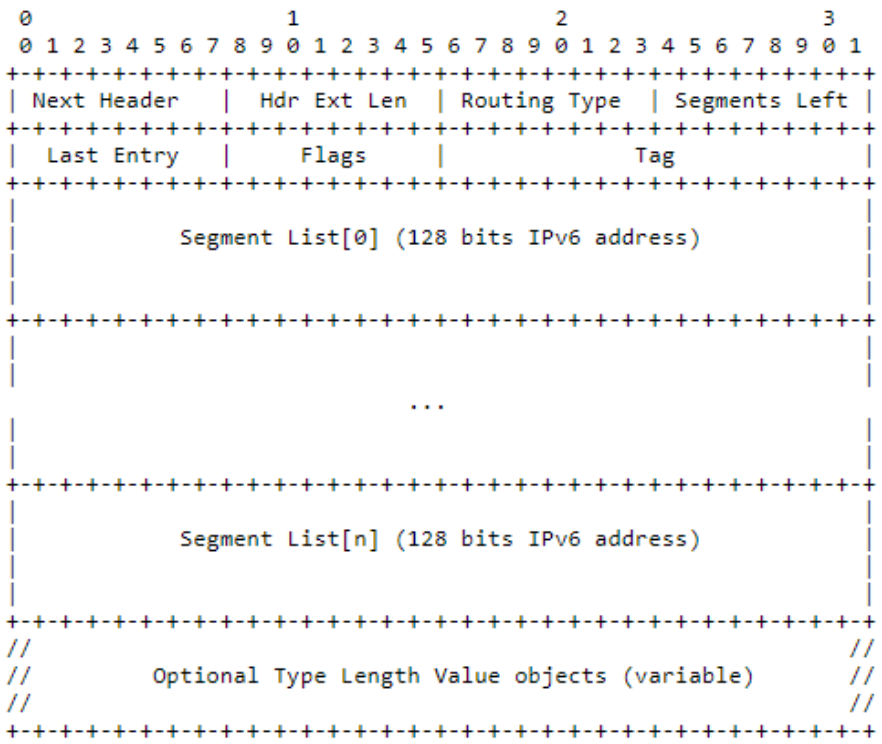


Figure 1.1: *Segment Routing Header.*

- **SRv6 Endpoint node**: receives an SRv6 packet where the destination address of that packet is (locally) configured as a SID on the node itself. In other terms, this node is where the SRv6 segment is terminated. The SRv6 Endpoint node executes the function bound to the SID. The most basic function of the SRv6 Endpoint node consists in processing the SRH by considering the Segment Left value. If the Segment Left is equal to zero, then it continues to process the next header in the packet. Conversely, the Segment Left is decreased by one unit. The destination address of the IPv6 header is updated with the SID pointed by the Segment Left. The Hop Limit is decremented and the packet is routed again, considering the new destination address.

1.2.3 Network Programming Model

The SRv6 Network Programming Model (NPM) is defined in [27] and extends the SRv6 architecture. The core idea of the SRv6 Network Programming Model consists in encoding, within the *Segment List*, instructions other than locations thanks to the huge IPv6 addressing space. Such instructions are executed by SRv6 nodes when SRv6 packets are received. Each SRv6 node handles one or more tables (SID tables) containing SIDs which are local to the node. For each entry into the SID table there is an associated function (i.e. *behavior*)). By combining functions distributed on different nodes, it is possible to achieve *a networking objective that goes beyond mere packet routing*. The Network Programming Model offers the possibility to implement virtually any complex service by combining the basic behaviors in a *network program* that is embedded in the SRH.

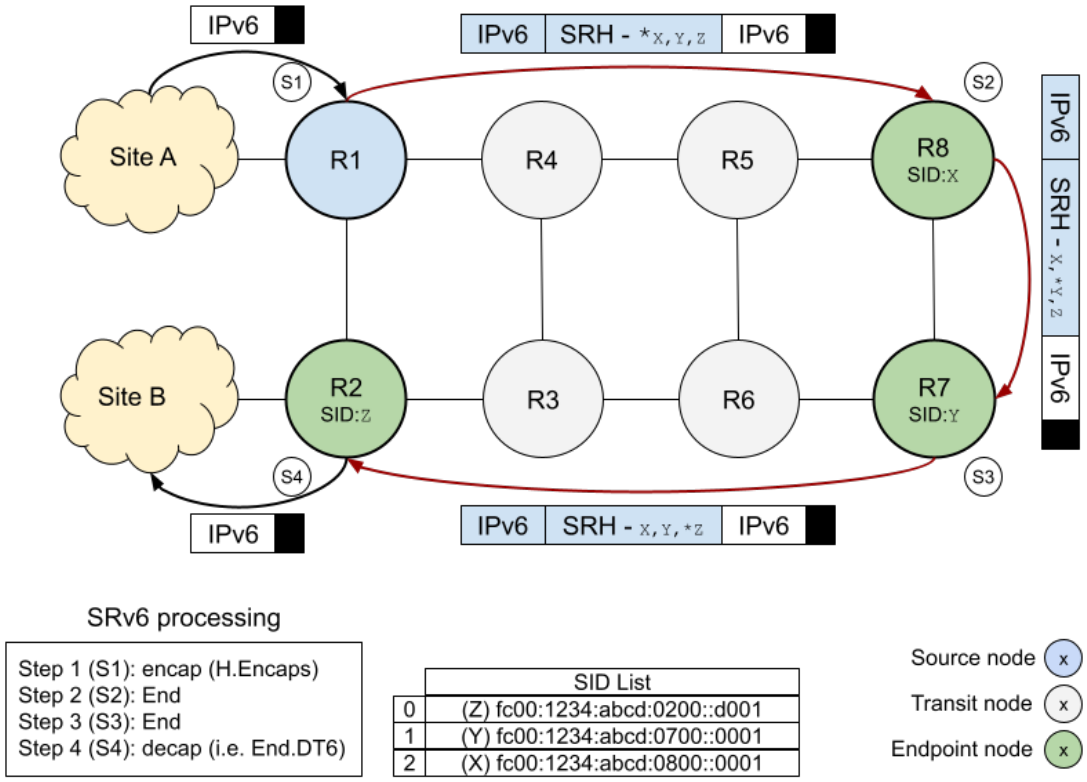


Figure 1.2: *SRv6 network programming in action.*

In the Network Programming Model, an SRv6 SID can be partitioned in three parts and expressed as LOC:FUNCT:ARG (Locator, Function, Argument). The Locator part can be routable and used to forward a packet to a specific node where a behavior, identified by the Function part, needs to be executed. In most cases, the Argument part is not used, hence a SID can be simply decomposed in two parts LOC:FUNCT (Locator and Function).

For example, an operator can use a /48 IPv6 network prefix (referenced as *Locator Block*) for its SRv6 transport domain which includes all SRv6 capable transport nodes. Each SRv6 capable node can be assigned a different /64 IPv6 network sub-prefix inside the Locator Block, therefore up to $2^{16} = 65356$ SRv6 nodes can be supported in this specific configuration. Inside each SRv6 node, 2^{64} different SIDs can be supported. With reference to Figure 1.2, the /48 Locator Block prefix can be `fc00:1234:abcd::/48`, a node prefix can be `fc00:1234:abcd:N::/64`, the SID of a behavior to be executed in the node can be `fc00:1234:abcd:0700::S`. Following such a schema, the Locator part (LOC) is represented by the leftmost 64 bits, representing the Locator Block and the node part N. In the example, the Locator for node R_N is `fc00:1234:abcd:0N00`. The FUNCT part is represented by the rightmost 64 bits (no ARG is considered). In the example, `0001` or `d001` are used (preceded by 12 more leading zeros in hexadecimal notation).

To ease the interoperability, the SRv6 Networking Programming Model defines a set of behaviors that can be bound to an SRv6 SID. SRv6 behaviors can be categorized in two distinct categories: *Headend* and *Endpoint* behaviors.

SRv6 Headend behaviors

One or more SRv6 Policies are often configured in SRv6 Source nodes, where the SRv6 encapsulated packets are originated. Each SRv6 Policy [148] includes a SID List that identifies a specific path across the network. SRv6 Headend[1] behaviors steer received packets into an SRv6 Policy. When an SRv6 Source node receives packets matching a configured SRv6 Policy, the Headend behavior associated with that SRv6 Policy is applied to those packets. SRv6 Headend behaviors (*modes*) are defined in [27] and below is the definition of the two most used:

- **H.Encaps**: is the Headend behavior with the encapsulation in an SRv6 Policy. In the encap mode, the original layer-3 packet (IPv4, IPv6) is carried as the inner packet of an IPv4/IPv6-in-IPv6 encapsulated packet. The outermost IPv6 header is followed by the Segment Routing Header (SRH) containing the SRv6 Policy (encoded as the Segment List);

- **H.Encaps.L2**: is the Headend behavior with the encapsulation of layer-2 frame in a SRv6 Policy. It works in the same way as the H.Encaps, but it considers the whole layer-2 frame rather than the layer-3.

SRv6 Endpoint behaviors

For each SID configured in an SRv6 Endpoint, there is a function to be executed on packets whose active segment matches that SID. Well-known functions (behaviors) that can be associated with the segments are defined in [27], among which the most widely used are:

- **End**: is the endpoint function. The Segment Left of SRH is decreased and the IPv6 destination address is replaced with the next active segment. The Forwarding Information Base (FIB) lookup is performed on the updated IPv6 destination address and the packet is forwarded accordingly. This behavior represents the SRv6 instantiation of a prefix SID;

- **End.X**: is the endpoint function with l3-layer cross-connect to an IPv6 adjacency representing the SRv6 instantiation of an adjacency SID. It is a variant of the End behavior where the FIB lookup is carried out on the adjacency SID rather than the IPv6 destination address of the packet;

- **End.T**: is the endpoint function with specific IPv6 table lookup. It is a variant of the End behavior where the FIB lookup is carried out on a specific routing table associated with the SID rather than the main routing table;

- **End.DT4**: is the endpoint function with decapsulation and specific IPv4 table lookup. It is mostly used for implementing IPv4 L3 VPNs in multi-tenant environments. This behavior is a variant of the End function where the inner IPv4 packet is depcauslated (removing the outer IPv6 and all the extension headers) and the FIB lookup is performed considering the IPv4 routing table associated with the SID;

[1] SRv6 Headend behaviors defined in RFC8986 were previously known as Transit behaviors in Network Programming Model drafts.

- **End.DT6**: is the endpoint function with decapsulation and specific IPv6 table lookup. It is identical to the End.DT4 but it considers inner IPv6 packets rather than IPv4 ones. Hence, after the decapsulation, the FIB lookup is performed on the IPv6 routing table associated with the SID;

It is worth noting that custom and advanced network functions such as NAT, firewall, DPI, etc. can be assigned to specific SIDs. In other words, there is no limitation on what behaviors/functions might be associated with SIDs carried in the Segment List. Segments can be composed together and encoded into Segment Lists that can assume the form of network programs.

A simple example

Figure 1.2 depicts a simple IP network which has been *programmed* considering the Network Programming Model. The purpose is to enable *Site A* and *Site B* to communicate with each other through a SR domain where some type of packet processing can be applied on traffic, for instance: packet inspection, firewall, VPNs and many more.

An IPv6 packet originated from *Site A* enters in the SRv6 domain (made of all the Nodes R_i such that $i = 1..8$) and is intercepted by router R_1. The router R_1 applies the SRv6 Policy by encapsulating (H.Encaps) the received IPv6 packet in a new IPv6 packet, followed by the SRH whose SID List corresponds to `X, Y, Z`. This results into a source routing policy that routes the packet through the path $R_8 \rightarrow R_7 \rightarrow R_2$, respectively identified by the SIDs `X`, `Y`, `Z` and then executes the *decap* operation. Thus, R_1 forwards the SRv6 packet to R_8. The packet will cross R_4 and R_5 which are, in this case, configured as SRv6 Transit nodes enforcing the "base" IPv6 forwarding. Once R_8 receives the packet, it applies the SRv6 End behavior because the active segment corresponds to the SID associated to that behavior. The packet gets processed (IPv6 DA is set to `Y`, the Segment Left (SL) is 1) and sent to R_7 which applies the same operations performed in R_8. Upon completion of the End behavior by R_7 (IPv6 DA is set to `Z`, SL is 0), the packet is forwarded to R_2 via $R_6 \rightarrow R_2$. Node R_2 is the last SRv6 router in the path, the IPv6 DA (active segment) of the packet matches the configured local SID that triggers the SRv6 End.DT6 behavior: it decapsulates the inner IPv6 packet, performs the lookup using the IPv6 routing table associated with the SID and forwards the packet to *Site B*.

This simple example shows the strength and potential of the Network Programming Model: behaviors (functions) run on each node could carry out any kind of processing which may alter the packet in any way. For instance, SR Policies could be used for monitoring specific traffic flows, performance and packet loss in a SRv6 network as I will cover in next Chapters.

1.2.4 Extending the SRv6 Network Programming Model with Micro SID

Micro SID is a straightforward extension to the SRv6 network programming model for the efficient representation of Segment Identifiers (SIDs). The fundamental idea of the Micro SID solution [28] is that each 16-byte instruction (SID) of an SRv6 packet can carry a micro-program, composed of micro-instructions represented with identifiers called Micro SIDs. A micro-instruction is represented with a *Micro SID*, also called *uSID*. Considering the most common configuration, Micro SIDs are represented with 2

Plain SRv6	Micro SID
End	uN
End.X	uA
End.DT4/End.DT6/End.DT2	uDT
End.DX4/End.DX6/End.DX2	uDX

Table 1.1: *Plain SRv6 behaviors and Micro SID behaviors.*

bytes, and up to 6 Micro SIDs can be carried in a regular 16 bytes SID. For example, an SR path of 5 hops would require 16*5=80 bytes to be represented as a regular "SR path", while it would require 16 bytes to be represented as a sequence of 5 Micro SIDs encoded in a single regular SID. Hence, the Micro SID approach results in a large saving of the packet overhead when multiple segments (instructions) need to be transported in an SRv6 packet.

As discussed in [181], some application scenarios for SRv6 may require long sequences of SIDs to be carried in the SRH packet header (e.g. up to 15 SIDs). In the current SRv6 model, this requires $N * 16$ bytes to be carried in the SRH, where N is the number of SIDs in the SID List. To this regard, Micro SID turns out to be a viable solution since it allows to reduce the SID List size, saving valuable space in the packet header without necessitating any changes to the SRH data plane. As described in [28], the Micro SID solution proposes to extend SRv6 Network Programming with new behaviors, called uN, uA, uDT, uDX and reported in Table 1.1.

To understand how the Micro SID operates, it is useful to provide a simple example based on the same SRv6 network topology already studied in Subsection 1.2.3. For the sake of clarity, in Figure 1.3 I reported the network topology and I outlined, step-by-step, the ongoing processing based on the Micro SID extension of the network programming model. In this example, a /32 prefix is chosen as Locator Block for the Micro SIDs (referred to as *uSID block*). All routers in the topology are assigned a /48 prefix from such as Micro SID block: `fcbb:bbbb::/32`. The ingress router R1 applies the Micro SID policy by encoding the address `fcbb:bbbb:0800:0700:0200:f00d::` into the IPv6 destination address of the outer IPv6 header (no SRH is needed at all, in this case). This results into a source routing policy that routes the packet through the path $R_8 \rightarrow R_7 \rightarrow R_2$, respectively identified by the Micro SIDs `0x0800`, `0x0700`, `0x0200` and then executes a *decap* operation. Thus, R_1 sends the packet to R_8. The packet will cross R_4 and R_5 that in this case enforce "base" IPv6 forwarding. As soon as R_8 receives the packet, the IPv6 destination address of the outer IPv6 header matches with the Micro SID assigned to the SRv6 uN behavior. It "consumes" its Micro SID identifier in the destination address: (i) the `0x0800` Micro SID is popped from the destination address; (ii) the remaining Micro SID list is left-shifted by 16 bits; (iii) the End of Container identifier (`0x0000`) is inserted in the last 16 bits. The resulting IPv6 destination address is `fcbb:bbbb:0700:0200:f00d::`. Upon completion of the procedures above, the packet is transmitted to R_7 which performs an analogous set of procedures that ends with the transmission of a packet containing the Micro SID List `fcbb:bbbb:0200:f00d::` to R_2 via $R_6 \rightarrow R_3$. Since R_2 is the last SRv6 router in the path, the destination address of the packet matches the FIB entry with destination `fcbb:bbbb:0200:f00d::/64`. This rule includes the terminator Micro SID `f00d` which triggers the final SRv6 End.DT6 behavior: the packet is decapsulated and

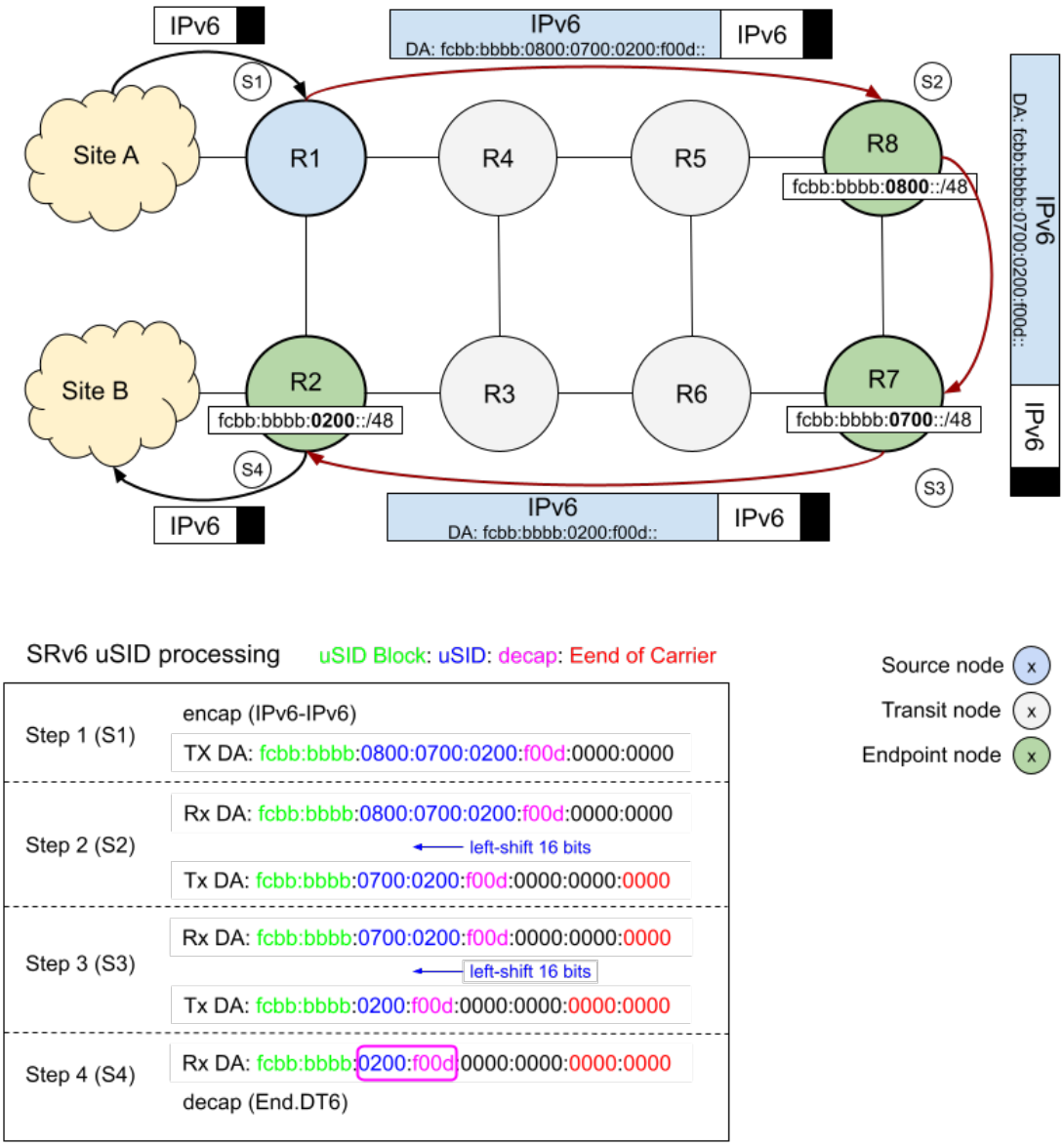


Figure 1.3: *Extending the SRv6 network programming model with Micro SID.*

handled by a specific IPv6 routing table.

The Micro SID solution fully leverages the SRv6 network programming solution. In particular, the data plane with the SRH dataplane encapsulation is leveraged without any change; any SID in the SID List can carry micro segments. As for the Control Plane, the SRv6 Control Plane is leveraged without any change. The mechanisms for the compression of SID identifiers are described in [26].

The Micro SID solution enables ultra-scale deployments (e.g. as needed for multi-domain 5G scenarios) and reduces the overhead at the minimum reducing the potential issues with MTU. It is fully compatible with SRv6 architecture, so it can run in mixed scenarios where only a subset of nodes support the Micro SIDs.

1.3 The Research on Open SRv6 Ecosystem (ROSE) project

Segment Routing over IPv6 (SRv6) Network Programming enables network operators, cloud providers or even applications to specify a packet processing program by encoding a sequence of instructions in the IPv6 packet header. In order to exploit the Network Programming Model to its full strength, it is appropriate to have a network infrastructure and nodes interconnected to each other. In this way, network programs can be written so that instructions are executed on the various nodes taking full advantage of SRv6 technology.

The ROSE (Research on Open SRv6 Ecosystem) project [137] addresses multiple aspects of the SRv6 technology: Data plane, Control plane, SRv6 host networking stack, integration with applications, integration with Cloud/Data Center Infrastructures. The ROSE project builds up and maintains a Linux-based Open Ecosystem for SRv6 which is composed of many sub-projects.

1.3.1 ROSE Sub-projects

The complete list of ROSE projects is available at [137], where hyperlinks are provided to access code, documentation and results. Several of the projects being part of

The topics I have been researching and described in this thesis are the backbone of many of the projects proposed in ROSE, such as:

- **SRPerf**: is an open source performance evaluation framework for software and hardware implementations of SRv6. I will discuss about the SRPerf framework and my contributions in Chapter 3;
- **SRNK - SR proxy Native Kernel**: is a SR proxy which acts a as relay mechanism in order to support SRv6 un-aware Virtual Network Functions. In Chapter 4, I will explain both the theory behind the concept of SR Proxy applied to IPv6 networks and the way in which I designed, implemented this component within the Linux kernel networking stack;
- **SRv6 uSID (Micro SID)**: is an extension to the SRv6 Network Programming Model and it allows expressing SRv6 segments (SIDs) in a very compact and efficient representation. In Chapter 2, I will describe how I realized the Micro SID solution in the Linux kernel, while I evaluate the performance of the provided implementation in Chapter 3;

- **SRv6-PM (SRv6 Performance monitoring)**: is an accurate Per-Flow Packet Loss Monitoring (PF-PLM) in Linux kernel in an SRv6 network. This project demonstrates how to leverage the Network Programming Model to program SRv6 nodes, controlled by an SDN/SR controller, for marking and counting packets in end-to-end communications among nodes. In Chapter 5, I will study the theory of performance monitoring applied to SRv6 networks and discuss how I designed and realized several implementations of PF-PLM within the kernel by exploiting different packet processing technologies;
- **HIKe-v0 - Hybrid In Kernel eBPF**: is an efficient programmable data plane architecture for Linux-based routers designed for supporting Hybrid SDN and SRv6 networks. HIKe-v0 integrates conventional Linux kernel networking processing with custom extended Berkeley Packet Filtering (eBPF) programs to accelerate and speed up performance of SRv6 software routers. In Chapter 6, I will discuss in depth the advantages of HIKe-v0, how it could be used in the HDSN/SRv6 architecture, how I designed and implemented this programmable data plane solution;
- **HIKe-v1 - Hide, Improve and desKill eBPF**: is an important improvement of the HIKe-v0 architecture which aims to simplify the writing of network applications using eBPF. In Chapter 7, I will explain how HIKe-v1 helps inexperienced eBPF programmers concatenate already pre-fabricated eBPF programs to achieve complex packet processing without going crazy with all the quirks of the eBPF world.

1.3.2 The ROSE SRv6-VM

The ROSE project comes with a ready-to-go Virtual Machine (ROSE SRv6-VM, for short) that is used for tutorial, development and testing purposes. Projects discussed in 1.3 can be run inside the ROSE SRv6-VM by following tutorials available at [137]. Indeed, the VM is shipped with several tools allowing the user to set, configure and emulate both the control plane and data plane of complex networks.

Mininet based emulation

The ROSE SRv6-VM includes an emulated network environment based on Miniet [84], [66] and relies on the Linux kernel implementation of the SRv6 data plane that I extended and improved during my research activities. Mininet is a network emulator used to create virtual network nodes such as virtual hosts, switches, controllers and (network) interfaces leveraging the Linux kernel networking stack. The separation among virtual network nodes is achieved using the Linux network namespaces. Linux network namespaces are a feature designed to isolate network environments through virtualization achieved at OS level. For example, using network namespaces, each process can create separate network interfaces and routing tables that are isolated from the rest of the system and operate independently. In Mininet, each virtualized host is represented by a spawned process that is created with its own networking stack. Therefore, virtualized hosts are able to communicate with each other through virtual ethernet pairs and switches. Using the Network namespaces, Mininet is able to run many virtual hosts, routers and switches on a single machine (laptop or PC).

Mininet comes with a command line (CLI) which is similar to that of the Linux command line, using it a user can interact with the system. Through the CLI, it is possible to type commands for displaying Mininet nodes (mininet hosts), links connecting nodes and other info regarding the network topology.

Actually, there is no exact user interface for the Mininet because everything in the Mininet is controlled either by Python scripts or commands. Therefore, an user can define its network topology using the offered Python API and then launch the script creating the virtual nodes, links, switches and so on. To this regard, ROSE defines several types of node such as SRv6 enabled routers, customer hosts, providers hosts and SDN controllers.

Once the virtual network has been set up by a Mininet script, each node has to be configured. There are two possibilities: i) the configuration of each single node is carried out in the same main script that created the network using the Python API; ii) each node is configured using bash script called in the main python script. ROSE subprojects adopt (ii) since it allows to keep the creation of networks networks separated from the configuration of each node. Doing so, configuration scripts can run on real nodes (SDN controllers, routers or hosts) and, thus, users can move the code developed on the Mininet testbed to a real system with minimal changes.

Control plane emulation

The ROSE ecosystem includes different control plane functionalities [150] of a Software Defined Network (SDN), including the management of SRv6 tunnels, exporting the network topology to a database and monitoring the network performance. The building blocks of the control plane are, fundamentally, the SDN controller and the Node Manager.

The SDN controller uses a gRPC [64] API for connecting to network nodes. The gRPC is a open source high-performance RPC framework which enables to define services using Protocol Buffers [41]. Serialized data is sent/received over TCP connections so that the SDN controller is able to interact with nodes and to enforce or retrieve configurations such as SRv6 paths or behaviors. Each node must provide an user space agent, named Node Manager, which connects to the SDN controller and receives commands or serialized data structures. The Node Manager is in charge of adapting the commands sent by the SDN controller (defined by the Southbound API) to configure the underlying networking stack (i.e. using `iproute2` tool suite).

The control plane supports different functionalities that are exposed to the applications through a Northbound API, including the setup of SRv6 paths, behaviors and tunnels, extracting the network topology from a set of nodes running dynamic routing protocols (i.e. ISIS, OSPF). In addition, a CLI is provided to offer the user the possibility for interacting with the SDN controller and pushing/pulling custom configurations.nd push/pull custom configurations.

1.3.3 The Linux playground

Linux offers massive support for both networking and virtualization. In fact, a single Linux-based operating system can host multiple virtual network nodes through both system virtualization (i.e. containers) and virtual machines. Regardless of the virtualization solution, a dedicated protocol stack is created for each virtualized entity that

is independent of the others. A Linux-based router thus becomes an extremely powerful and versatile networking tool. Developing solutions for the Linux kernel or more generally for Linux-based network devices (i.e. routers) allows designers to deploy their solutions on a wide range of devices ranging from embedded devices, personal computers to High Performance Computing (HPC) clusters.

CHAPTER 2

The Network Programming Model in the Linux kernel

The Linux kernel is one of the largest and most popular open source projects in the world [90]. It was designed in the early nineties and since then it has experienced a non-stop evolution, becoming one of the most popular kernels in the world with a massive hardware support. Not surprisingly, operating systems based on the Linux kernel are employed ranging from embedded systems to supercomputers. Currently, the Linux kernel has more than 24 million lines of code and about 1600 developers [89]. Although the open source community around Linux provides several resources and documentation describing the inner workings of the kernel, the learning curve in developing new features is very steep. This is also due to the amount of documentation that is, unfortunately, not always up-to-date.

Providing a detailed description about how the Linux kernel networking stack works is beyond the scope of this thesis. For an in-depth study regarding the Linux kernel networking internals, I recommend consulting [138]. In the later sections of this chapter I address, in reasonable detail, the implementation aspects of the IPv6 networking and SRv6 subsystem in the Linux kernel.

In Section 2.1, I briefly introduce the main concepts about the Linux kernel networking. I do not pretend to be exhaustive since the networking stack is a very complex system. In Section 2.2, I describe in more detail how I extended the Linux SRv6 Network Programming Model implementation, highlighting some of my principal contributions. In Section 2.3, I provide an overview of the extended Berkeley Packet Filter (eBPF) necessary for understanding the topics discussed in the next Chapters. Finally in Section 2.4, I consider alternative implementations to the SRv6 Linux kernel subsystem which rely on a custom kernel module and user space packet processing solutions.

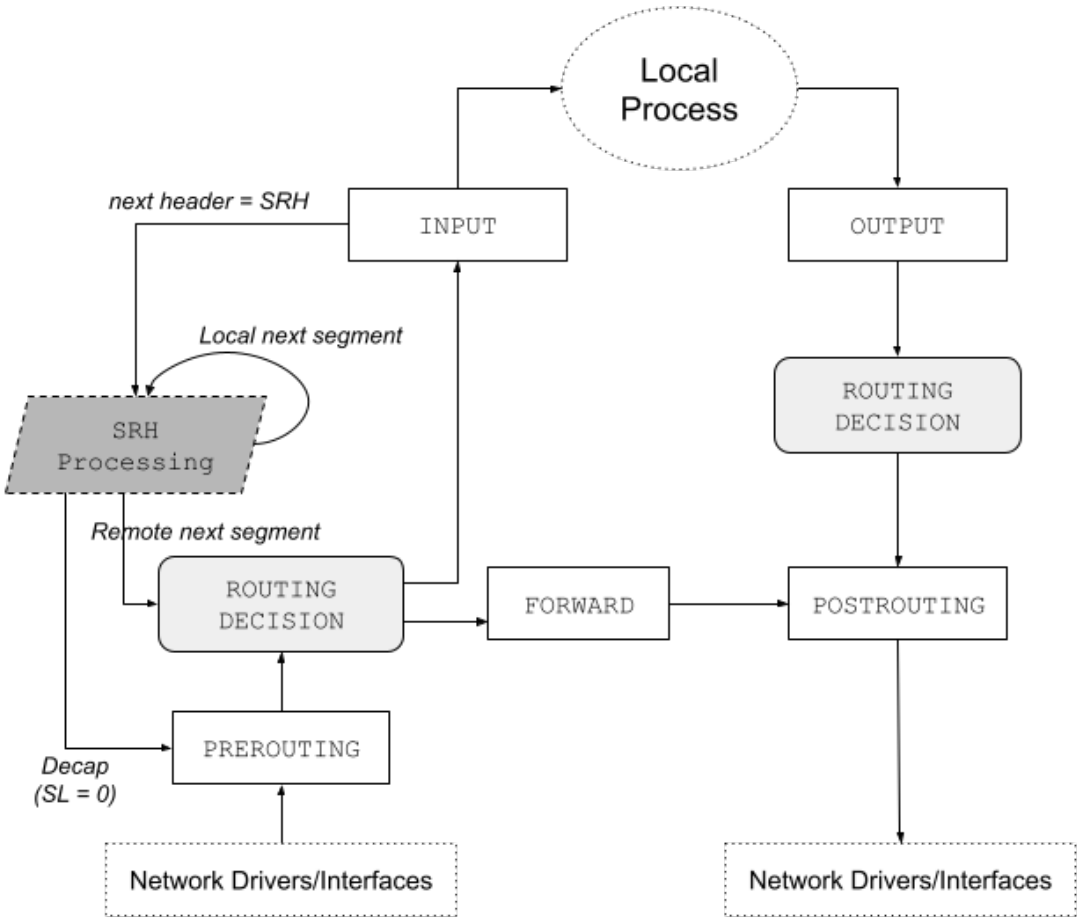


Figure 2.1: *High level overview of the IPv6 Linux kernel (v4.10) networking stack where processing stages are highlighted.*

2.1 Linux kernel networking subsystem

2.1.1 Packet processing in the IPv6 networking stack

The Linux kernel networking is a very complex system and implements dozens of networking protocols at its core. In the kernel, a packet is represented as a socket buffer or `skb` for short. An `skb` is described through a specific data structure of type `struct sk_buff` that contains the on-the-wire received data packet as well as some metadata. Such metadata includes, for example, the ingress interface where the packet was received, header offsets and useful information needed for processing. The `sk_buff` structure is used practically everywhere in the networking stack and for this reason there are literally tens of helper functions that provide the manipulation of the `skb`.

From an architectural point of view, the networking stack is divided into several layers ranging from the more hardware-oriented network drivers up to the socket APIs used for interacting with the user interface.

Figure 2.1 shows, at a very high level, the main IPv6 traffic processing stages occurring within Linux kernel networking stack. Each packet received from the network card driver is stored within the data area of a `skb` and analyzed to determine how it should

be processed. Once identified as an IPv6 packet, it is forwarded to the IPv6 layer. The first stage of processing is represented by PREROUTING in which several basic sanity checks are performed on the packet. Next, the routing decision is taken on the IPv6 destination address (IPv6 DA) using the longest prefix match algorithm. The routing process involves one or more routing tables that have been populated by the control plane and the outcome of the routing decision can basically lead to different processing stages:

i the packet is meant for local processing since the IPv6 DA address matches one of the local addresses associated with a network interface. Therefore, the skb is delivered to the INPUT stage for processing upper-layers such as TCP and UDP;

ii the packet is meant to be forwarded to another network node. Thus, the skb is delivered to the FORWARD stage which is in charge of verifying the presence of proxies, decrementing the hop limit, etc. Subsequently, the skb processing enters in the POSTROUTING stage where the resolution of the next-hop is done and the packet is handled by low-lever networking layers before being sent out of the network card.

Packets which are locally generated by the node (i.e. applications that send TCP, UDP traffic) arrive directly at the OUTPUT stage. Then, a routing operation is performed using the IPv6 DA to determine the egress interface. At this point, the processing continues in the POSTROUTING stage in the same way as described in the FORWARDING one.

Starting with version 4.10 of the Linux kernel, the processing pattern described so far has been extended with Segment Routing support for IPv6. With this extension, an IPv6 packet with SRH that arrives at the configured SR endpoint node is processed in the INPUT stage since the IPv6 DA is bound to a local address. Therefore, two basic actions are performed in the INPUT on the basis of the Segment Left (SL) field contained in the SRH:

- SL > 0, the packet must be updated to the next segment; this is done by decrementing the value of the SL and updating the IPv6 DA with the segment (present in the Segment List) indicated by the SL. This operation is repeated until the segment is no longer local to the node. At this point, the skb is subjected, once again, to the routing process in which the packet fate decision is made based on the updated destination address;

- SL = 0, with SRH following the IPv6[1] header. In this case, the internal packet is extracted and the packet processing restarts from the PREROUTING stage.

However, this extension is very limited indeed. There is no possibility for inserting the Segment Routing Header into existing traffic nor performing any behavior other than the End one. For these reasons, further extensions to SRv6 processing have been proposed and adopted in the kernel and they are based on the Lightweight Tunnel (LWT) infrastructure [79].

[1] Support for inner IPv4 header has been introduced in a later patch.

2.1.2 SRv6 Headend behaviors in the Linux kernel

Starting from Linux kernel 4.10, SRv6 Headend behaviors (formerly known as SRv6 Transit behavior) such as H.Encaps, H.Insert have been implemented using the the Lightweight Tunnel (LWT) infrastructure [79]. LWTs are a technique for implementing interfaceless tunnels. Each route in a kernel routing table is associated with two function pointers, respectively `input` and `output`. The purpose of these functions is to determine how the packet should be processed after the Forwarding Information Base (FIB) lookup. The `input` function is invoked in the IPv6 receiving path after the routing decision to establish whether the packet should be processed in the `INPUT` or `FORWARD` stage. Similarly, the `output` function is invoked in the IPv6 transmission path immediately after the routing decision determining how the packet should be transmitted over the network. For an IPv6 route, the `output` function pointer references the `ip6_output()` that transmits the `skb` to the egress interface. The `input` function pointer depends on whether the route is local or non-local. If the route is local, the pointer is set to `ip6_input()` while it is set to `ip6_forward()` otherwise. The core idea of LWTs consists of overriding those function pointers with custom `input` and `output` functions to change the way packets are going to be processed. For implementing a lightweight tunnel, one needs to specify `input` and/or `output` processing functions. In addition, a developer can also define additional functions that aim to manage per-tunnel stateful data to be saved in the route. Accordingly, a kernel route created to make use of a given LWT will have `input` and/or `output` function pointers referring to those of the specific tunnel type.

The LWT infrastructure named `seg6` has been implemented to support SRv6 Headend behaviors [87] such as H.Encaps and H.Insert. Using the `rtnelink` protocol, an user can create IP/IPv6 routes and specify the `seg6` LWT processing. In this way, the `seg6` lightweight tunnel replaces both the `input` and `output` functions pointing to the `seg6_do_srh()` where the SRH processing happens.

The user space `iproute2` tool-suite, which implements the `rtnetlink` protocol, can be used for instantiating new SRv6 Headend behaviors as shown in Listing 2.1.

Listing 2.1: *`iproute2` command to create a new instance of SRv6 End.DT4 behavior.*

```
$ ip route dead:beef::/64 encap seg6 mode encap segs fc00::1,fc01::1 dev eth0
```

2.1.3 SRv6 network programming model in the Linux kernel

The core idea behind the SRv6 network programming model is that each SR node should configure a *My Local SID* table where each entry is a segment mapped to a processing function. When a packet enters the SR node with an active segment matching an entry of the *My Local SID*, the associated function is applied to the packet.

The `seg6local` LWT infrastructure [86] has been implemented to support the SRv6 network programming model in the Linux kernel since version 4.14 [93]. This new LWT type is designed to meet the requirements imposed by the networking model principles. Specifically, the `seg6local` LWT enables i) the processing of a packet that may have a valid active segment not assigned to any interface of the SR node; ii) the processing of a packet that does not necessarily come with the SRH.

The `seg6local` LWT infrastructure[2] aims to replace the `input` function pointer of a route with the `seg6_local_input()`. Each `seg6local` LWT instance makes use of the `action` attribute[3] to identify the SRv6 Endpoint behavior to be applied on the packet and as well as a possible set of mandatory attributes which may be required for some specific behaviors. When a kernel route is created with the `seg6local` LWT, the tunnel state represented by `struct seg6_local_lwt` is stored within the route itself. This data structure contains several fields which aim to keep track of the specific SRv6 Endpoint behavior function to be applied on packets, the set of mandatory attributes for that behavior as well as the needed space for storing them.

In Linux kernel version 4.14, there are nine implemented SRv6 Endpoint behaviors. In kernel version 4.17 [94], a new SRv6 Endpoint behavior named End.BPF [182] was introduced. This Endpoint behavior does not exist in the SRv6 network programming model [31] but is a peculiar feature of the Linux kernel. Indeed, the End.BPF allows an user to instantiate an SRv6 Endpoint that i) advances the SRH [4] to the next segment; ii) executes a processing function whose logic is given by the associated eBPF (extended Berkeley Packet Filter) [44] program. eBPF is a general-purpose 64 bits RISC-like virtual machine included in the Linux kernel. An eBPF program can be written in restricted C and compiled to eBPF bytecode using LLVM [128]. Then, such bytecode is attached to some predefined hooks in the kernel to dynamically add new features and capabilities to it. The End.BPF can be very useful since it allows users to create, apply new custom processing functions on SRv6 traffic such as load balancing and fast rerouting without the hassle of re-compiling the kernel. However, programs hooked to End.BPF have partial access to the SRH as well as to internal kernel functions which is achieved only through specific helper functions.

2.1.4 Policy Routing

Policy Routing extends the traditional routing based only on IP destination addresses. With Policy Routing the forwarding decision on a packet can be based on different features of the packets, considering packet headers at different protocol levels, incoming interfaces, packet sizes and so on. According to this extended routing model, the Linux kernel implementation of Policy Routing complements the conventional destination based routing table (that leverages the longest prefix match) with a Routing Policy DataBase (RPDB). In general, each *Policy Routing* entry in the RPDB consists of a selector and an action. The rules within the RPDB are scanned in decreasing order of priority. If the packet matches the selector of an entry the associated action is performed, for example an action can direct the packet to a specific routing table. The most important consideration for our purposes is that the RPDB rules are sequentially processed, so the performance penalty of checking the rules increases linearly with the number of rules.

2.1.5 Virtual Routing and Forwardng (VRF) infrastructure

A Virtual Routing and Forwarding (VRF) [177] device provides the ability to create virtual routing and forwarding domains (also known as VRFs) in the Linux kernel net-

[2] In this thesis, I also refer to the `seg6local` LWT infrastructure as SRv6 Endpoint behavior subsystem

[3] I prefer call a *parameter* with the name of *attribute* as it is referred in this way within the source code.

[4] the SRH must be present and the Segment Left must be > 0.

working stack. VRFs are especially useful in multi-tenancy problems where each tenant has their own unique routing tables and different routing gateways. A VRF provides traffic isolation at layer 3 for routing, similar to how a VLAN is used to isolate traffic at layer 2. Network devices can be enslaved to a VRF device and, at this point, local addresses and connected routes are moved to the table associated with that VRF.

Packets received on an enslaved device are switched to the VRF devices in the IPv4 and IPv6 networking stack giving the impression that such packets flow through the VRF device. On egress, packets are delivered to the VRF device driver before getting sent out the actual interface.

2.1.6 The Netfilter framework

The Netfilter framework [119] provides packet filtering, mangling, logging software for the Linux kernel. Netfilter consists of several hooks within the Linux kernel that allow modules to register callback functions at different locations of the kernel networking stack. Such registered callback functions are then invoked for every packet that traverses the respective hook in the networking stack. Currently, five hooks are supported in the kernel and they can be described as follows:

- `NF_INET_PRE_ROUTING`: provides the first Netfilter hook in the path of received packets. It allows to access received packets in the `PREROUTING` stage before being processed by the routing subsystem;
- `NF_INET_INPUT`: provides access to packets which are meant to be delivered/processed locally to the host. Thus, the hook is triggered when the routing subsystem has detected that a given packet should be handled locally (i.e. by an application) in the `INPUT` stage and it does not require to be forwarded to another host;
- `NF_INET_FORWARD`: provides access to packets that have to be forwarded to another host. The hook is activated in the `FORWARD` stage after the routing subsystem has determined that a packet is not intended to be processed locally and, therefore, requires to be sent to someone else on the network;
- `NF_INET_LOCAL_OUT`: triggered in `OUTPUT` stage and it provides access to locally generated outbound traffic;
- `NF_INET_POST_ROUTING`: provides access to packets at the `POSTROUTING` stage and before they are handled by the NIC to be sent over the wire.

As stated before, registered function callbacks are meant to apply different kinds of processing to packets. Hence, packets can be modified or dropped by the callback functions and the kernel must be informed about that. For this purpose, all callback functions in Netfilter have to return one of the possible actions defined hereafter:

- `NF_ACCEPT`: instruct the kernel to continue traversal of the packet normally;
- `NF_DROP`: stop the packet traversal through the networking stack and just drop it;
- `NF_STOLEN`: notify the kernel to forget about the packet since the callback is taking care of it;

- `NF_QUEUE`: instruct the kernel to push the packet into a given queue (usually for user space processing);
- `NF_REPEAT`: invoke again all the callbacks registered in the specific hook to process the packet one more time.

The Netfilter framework is incredibly extensible and several services can be built on top of it for achieving different packet processing operations. Indeed, using Netfilter a developer can write kernel modules to implement new network-oriented features without the need of recompiling the whole kernel. Implementation of several network functions such as NAT and firewall totally relies on the Netfilter framework. To this regard, `iptables` [74] is the user space utility program that allows a system administrator to configure packet filter rules for the Linux kernel firewall[5].

Through `iptables` it is possible to process SRv6 traffic since the firewall has been extended to handle SRH. To this end, I have extended the Netfilter/Xtables architecture to implement, on Linux-based software routers, different performance monitoring solutions tailored for SRv6 networks (see Chapter 5).

2.2 Extensions to the Linux kernel networking stack

In this Section, I am going to discuss some new brand features I have introduced into the Linux kernel networking stack with the goal of extending the implementation of the network programming model. I start by describing, in Subsection 2.2.1, the architectural changes made to the `seg6local` LWT infrastructure for supporting the *optional attributes* feature in Endpoint behaviors. Optional attributes enable to configure Endpoint behaviors so that they can operate in different ways depending on the set of attributes provided by the user space. In Subsection 2.2.2, I discuss improvements made to the `seg6local` LWT infrastructure to support optional per-behavior constructor/destructor callbacks which may be needed by SRv6 Endpoints during the setup and/or teardown of tunnel instances. In Subsection 2.2.3, I explain how I implemented the *Counters* as described in the network programming model specification. In Subsection 2.2.4, I describe the new *Strict Mode* concept that I introduced and implemented within the Virtual Routing and Forwarding (VRF) infrastructure. The Strict Mode is a core feature that allowed me to implement the SRv6 End.DT4 behavior in the Linux kernel and this activity is described in Subsection 2.2.5. In Subsection 2.2.6, I explain the way I improved the Linux SRv6 End.DT6 behavior implementation to also support the VRF-based operating mode. Then, the SRv6 End.DT46 behavior implementation is addressed in Subsection 2.2.7 where I give an idea of the benefits that such behavior brings in complex network scenarios. Finally in Subsection 2.2.9, I merely outline the problems and bugs I found while designing and implementing those kernel patches.

2.2.1 Optional attributes support for SRv6 Endpoint behaviors

In Linux, each SRv6 Endpoint behavior specifies a set (even empty) of required attributes that must be provided by the user space when that behavior is going to be instantiated. If at least one of the required attributes is not provided, the creation of the behavior instance fails.

[5]The entire firewall architecture is usually referred to as *Xtables*.

The `seg6local` LWT infrastructure lacks a way to manage optional attributes. By definition, an optional attribute for a SRv6 Endpoint behavior consists of an attribute which may or may not be provided by the user space. Therefore, if an optional attribute is missing (and thus not supplied by the user) the creation of the behavior instance goes ahead without any issue.

To support optional attributes, I extended the `seg6local` LWT infrastructure in order to differentiate the required attributes from the optional attributes. In particular, each behavior can declare a set of required attributes and a set of optional ones. The semantic of the required attributes remains **totally** unaffected by this change. The introduction of the optional attributes does **not** impact on the backward compatibility of the existing SRv6 Endpoint behaviors. To achieve such results, I modified the `struct seg6_action_desc` by adding a new field named `optattrs`. This `optattrs` is used for specifying all the optional attributes supported by a given behavior. In case one of the optional attributes is not provided (by the user space) during the creation of the behavior instance, then no error occurs. In `struct seg6_local_lwt`, I introduced a new field called `parsed_optattrs` to track the optional attributes that have been effectively parsed when a new behavior is instantiated. This field is needed whenever a behavior instance is about to be destroyed and the acquired resources associated with the optional attributes are released.

It is important to note that if an SRv6 Endpoint behavior receives an unexpected (optional or required) attribute, that behavior simply discards such attribute without generating any error or warning. This operating mode remains unchanged both before and after the introduction of the optional attributes extension.

Optional attributes are one of the key components needed to implement the SRv6 End.DT6 behavior based on the Virtual Routing and Forwarding (VRF) framework. Such attributes make possible the coexistence of the already existing SRv6 End.DT6 implementation with the new SRv6 End.DT6 VRF-based implementation without breaking any backward compatibility. Further details on the SRv6 End.DT6 behavior (VRF mode) are reported in 2.2.6.

From the user space point of view, the support for optional attributes *do not* require any changes to the user space applications, i.e. `iproute2` unless new attributes (required or optional) are needed.

2.2.2 Custom constructor/destructor callbacks for SRv6 Endpoint behaviors

An SRv6 Endpoint behavior may require some additional processing when a new instance is going to be created and/or destroyed. The SRv6 Endpoint subsystem lacks this capability and it does not enable the introduction of new complex behaviors that need to, for instance, allocate some memory just after having parsed the input attributes successfully. To overcome this limitation, I created a new data structure named `seg6_local_lwtunnel_ops` (shown in Listing 2.2) that contains two callbacks:

- `build_state()`: used for calling a custom constructor function during the initialization of the behavior instance and only after that all attributes have been parsed successfully;
- `destroy_state()`: used for calling a custom destructor function before the behavior instance is going completely destroyed.

Listing 2.2: *The definition of `seg6_action_param` structure used for customizing the creation/destruction of a behavior.*

```
/* callbacks used for customizing the creation and destruction of a behavior */
struct seg6_local_lwtunnel_ops {
        int (*build_state)(struct seg6_local_lwt *slwt, const void *cfg,
                           struct netlink_ext_ack *extack);
        void (*destroy_state)(struct seg6_local_lwt *slwt);
};
```

In the `seg6_action_desc` structure, I added the new `slwt_ops` field in order to link the callbacks defined in `seg6_local_lwtunnel_ops` to the behavior. In this way, during the definition of a behavior, a developer can easily specify the `build_state()` and `destroy_state()` callbacks to be invoked when a new instance of that behavior is going to be created and/or destroyed as exemplified in Listing 2.3.

Listing 2.3: *An example that shows how to define and use custom `build_state()` and `destroy_state()` callbacks during the creation/destruction of a behavior instance.*

```
...

static int new_behavior_build(struct seg6_local_lwt *slwt, const void *cfg,
                              struct netlink_ext_ack *extack)
{
    /* do something */
    ...
}

static void new_behavior_destroy(struct seg6_local_lwt *slwt)
{
    /* undo something */
    ...
}

static struct seg6_action_desc seg6_action_table[] = {
    ...
    {
        .action         = SEG6_LOCAL_ACTION_NEW_BEHAVIOR,
        .attrs          = ...,
        .input          = ...,
        .slwt_ops       = {
                            .build_state = new_behavior_build,
                            .destroy_state = new_behavior_destroy,
                          },
   },
   ...
};
```

I proposed this this solution in [152] and the patch was merged into the Linux kernel mainline since version 5.11 [91].

2.2.3 Counters for SRv6 Endpoint behaviors

I extended the the Linux kernel SRv6 Endpoint subsystem to implement *counters* defined in [31]. For each SRv6 Endpoint behavior, counters enable tracking of:

- the total number of packets that have been correctly processed;

- the total amount of traffic in bytes of all packets that have been correctly processed.

In addition, I introduced a new counter that counts the number of packets that have **not** been properly processed (i.e. errors) by an SRv6 Endpoint behavior instance.

Counters are not only interesting for network monitoring purposes (i.e. counting the number of packets processed by a given behavior) but they also provide a simple tool for checking whether a behavior instance is working as we expect or not.

Counters can be useful for troubleshooting misconfigured SRv6 networks. Indeed, an SRv6 behavior can silently drop packets for very different reasons (i.e.: wrong SID configuration, interfaces set with SID addresses, etc) without any notification/message to the user.

Due to the nature of SRv6 networks, diagnostic tools such as ping and traceroute may be ineffective: paths used for reaching a given router can be totally different from the ones followed by probe packets. In addition, paths are often asymmetrical and this makes it even more difficult to keep up with the journey of the packets and to understand which behaviors are actually processing our traffic.

When counters are enabled on an SRv6 Endpoint behavior instance, it is possible to verify if packets are actually processed by such behavior and what is the outcome of the processing. Therefore, the counters feature offer an non-invasive observability point which can be leveraged for both traffic monitoring and troubleshooting purposes.

Each SRv6 Endpoint behavior instance can be configured, at the time of its creation, to make use of counters. To accomplish this task, I extended the user space *ip* tool-suite allowing the user to create an SRv6 Endpoint behavior instance specifying the optional `count` attribute as shown in Listing 2.4.

Listing 2.4: *An SRv6 End behavior configured with counters turned on.*

```
$ ip -6 route add 2001:db8::1 encap seg6local action End count dev eth0
```

Per-behavior counters can be shown by adding "-s" to the *ip* command line as indicated in Listing 2.5.

Listing 2.5: *SRv6 End behavior counters shown using the ip command with the "-s" argument*

```
$ ip -s -6 route show 2001:db8::1

2001:db8::1 encap seg6local action End packets 0 bytes 0 errors 0 dev eth0
```

Counter support for SRv6 Endpoint behavior is realized through two different patches. The first patch [151] introduces the extensions to the Linux kernel and was merged into the mainline since version 5.13 [92]. The second patch [157] is applied to `iproute2` becoming part of the mainstream release.

2.2.4 Strict mode for the Virtual Routing and Forwarding (VRF) infrastructure

During the creation of a new Virtual Routing and Forwarding (VRF) device, it is necessary to specify the associated routing table used during the lookup operations. Currently, there is no mechanism that avoids creating multiple VRFs sharing the same

routing table. In other words, it is not possible to force a one-to-one relationship between a specific VRF and the table associated with it. To cope with those issues, I implemented in the Virtual Routing and Forwarding infrastructure of the Linux kernel a new functionality named *strict mode*.

The *strict mode* imposes that each VRF can be associated to a routing table only if that routing table is not already in use by any other VRF. In particular, the *strict mode* ensures that:

1. given a specific routing table, the VRF (if exists) is uniquely identified;
2. given a specific VRF, the related table is not shared with any other VRF.

Constraints 1) and 2) force a one-to-one relationship between each VRF and the corresponding routing table.

The *strict mode* feature is designed to be network namespace aware and it can be directly enabled/disabled acting on the `strict_mode` parameter. I decided to use the `sysctl` [103] command for enabling users to view and change the `strict_mode` parameter on-the-fly.

Read and write operations are carried out through the straightforward `sysctl` command on *net.vrf.strict_mode* path as shown in Listing 2.6.

Listing 2.6: *Turning on the strict mode kernel parameter using the `sysctl` command.*

```
$ sysctl -w net.vrf.strict_mode=1
```

Only two distinct values $\{0, 1\}$ are accepted by the `strict_mode` kernel parameter:

- with `strict_mode=0` (disabled), multiple VRFs can be associated with the same table. This is the (legacy) default kernel behavior, the same that users experience when the *strict mode* feature is not available in the kernel (i.e. old kernels that do not include the *strict mode* feature);
- with `strict_mode=1` (enabled), the one-to-one relationship between the VRFs and the associated tables is guaranteed. In this configuration, the creation of a VRF which refers to a routing table already associated with another VRF fails and the error is returned to the user.

The Linux kernel keeps track of the associations between a VRF and the routing table during the VRF setup, in the control plane. Therefore, the *strict mode* does not impact the performance or the intrinsic functionality of the data plane in any way.

When the *strict mode* is active it is always possible to disable the *strict mode*, while the reverse operation is not always permitted. Setting the `strict_mode` parameter to 0 is equivalent to removing the one-to-one constraint between any single VRF and its associated routing table. Conversely, if the *strict mode* is disabled and there are multiple VRFs that refer to the same routing table, then it is prohibited to set the `strict_mode` parameter to 1. In this configuration, any attempt to perform the operation will lead to an error and it will be reported to the user. To enable the *strict mode* once again (by setting the `strict_mode` parameter to 1), the user must first remove all the VRFs that share common tables.

There are several use cases which can take advantage from the introduction of the *Strict mode*. In particular, this new feature enables to:

i guarantee the proper functioning of some applications which deal with routing protocols;

ii perform some tunneling *decap* operations which require the use of specific routing tables for segregating and forwarding the traffic.

Considering i), the creation of different VRFs that point to the same table leads to the situation where two different routing entities believe they have exclusive access to the same table. By enabling the *strict mode* it is possible to prevent this situation which often occurs due to incorrect configurations done by the users. The ability to enable/disable the *strict mode* functionality does not depend on the tool used for configuring the networking. In essence, the *strict mode* feature solves at the kernel level what some other attempts such as [179] had tried to solve at the user space level (using only *iproute2*) with all the related problems.

Considering ii), the introduction of the *strict mode* functionality allows me to implement the SRv6 End.DT4 behavior. Such behavior terminates a SR tunnel and it forwards the IPv4 traffic according to the routes present in the routing table supplied during the configuration.

The SRv6 End.DT4 behavior can be implemented in the SRv6 networking subsystem by exploiting the routing capabilities made available by the VRF infrastructure. This behavior could leverage a specific VRF for forcing the traffic to be forwarded in accordance with the routes available in the VRF table. Anyway, in order to make the End.DT4 properly work, it must be guaranteed that the table used for the route lookup operations is bound to one and only one VRF. In this way, it is possible to use the table for uniquely retrieving the associated VRF and for routing packets.

The introduction of the *strict mode* for the Virtual Routing and Forwarding infrastructure has been considered interesting and relevant for the community so that patches [178] implementing this feature have been merged into the Linux kernel mainline since version 5.9 [95].

2.2.5 SRv6 End.DT4 behavior support in the Linux kernel

The SRv6 End.DT4 behavior defined in [31] is used to implement IPv4 L3VPN use-cases in multi-tenants environments. It decapsulates the received packets and it performs IPv4 routing lookup in the routing table of the tenant.

Support for the SRv6 End.DT4 behavior in the Linux kernel presents several challenges and some of them are particularly complex. The routing API for IPv4 in the Linux kernel is different from that of IPv6. Routing for IPv6 provides an API used to resolve routes within a specific routing table. However, this functionality is not currently present in the IPv4 routing subsystem. Modifying the API for supporting this need is not recommended in this case for a number of reasons. In general, the routing system is very complex and at the same time represents a fundamental element in the networking. The introduction of changes could lead to side-effects that are difficult to detect and they can create issues that are difficult to solve. Thankfully, the Linux

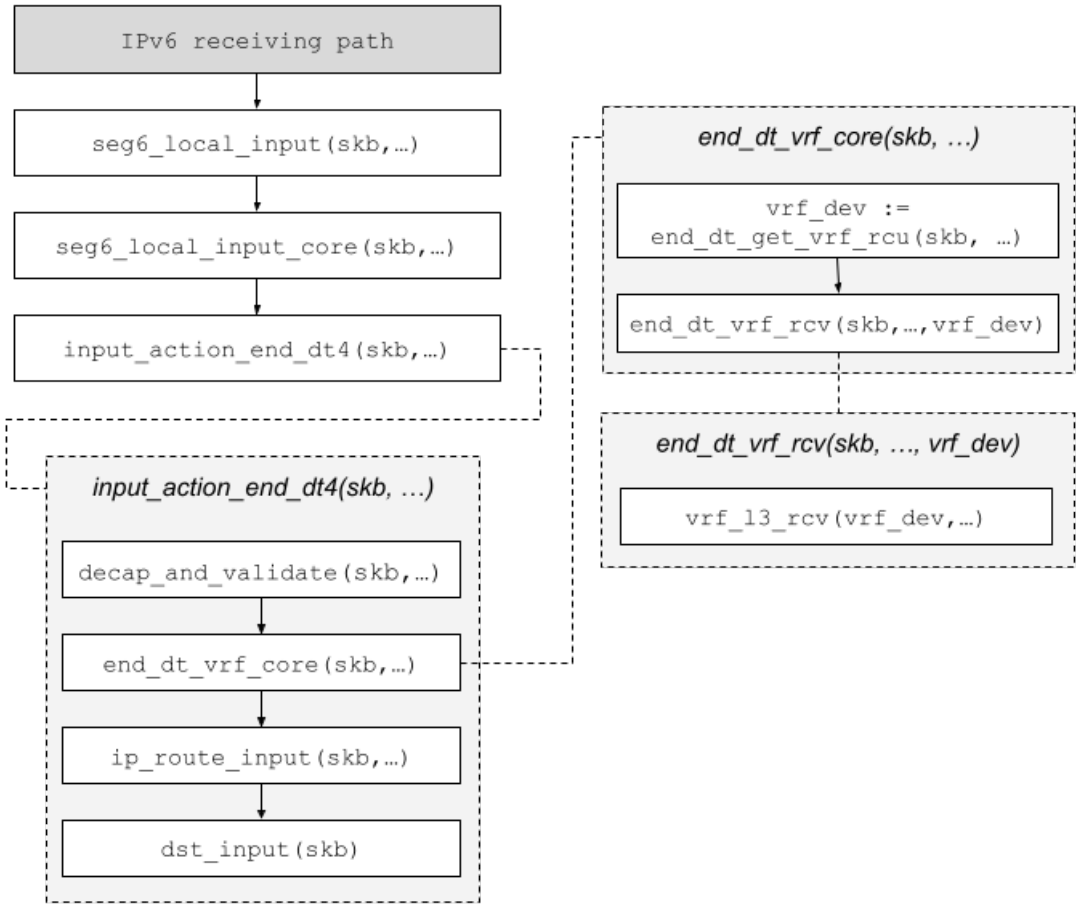


Figure 2.2: *SRv6 End.DT4 behavior receiving function in Linux kernel.*

kernel includes the Virtual Routing and Forwarding (VRF) infrastructure [177] which provides the ability to create virtual routing and forwarding domains in the Linux networking stack.

To implement the SRv6 End.DT4 behavior in the SRv6 Endpoint subsystem, I heavily exploited the VRF infrastructure. Indeed, the SRv6 End.DT4 implementation leverages a VRF device in order to force the routing lookup into the associated routing table. To make the End.DT4 work properly, it must be guaranteed that the routing table used for routing lookup operations is bound to one and only one VRF during the tunnel creation. Such constraint has to be enforced by enabling the VRF *sysctl* `strict_mode` parameter as shown in Listing 2.6. The new *sysctl* `strict_mode` kernel parameter is described in 2.2.4.

An SRv6 End.DT4 behavior instance can be created as shown in Listing 2.7

Listing 2.7: *`iproute2` command to create a new instance of SRv6 End.DT4 behavior.*

```
$ ip route add 2001:db8::1 encap seg6local action End.DT4 vrftable 100 dev eth0
```

The `vrftable` [154] is a new attribute introduced in `iproute2` for supporting the SRv6 End.DT4 behavior. In this case, the attribute is considered to be mandatory and it must be provided during the creation of a new SRv6 End.DT4 behavior instance.

The `input_action_end_dt4()` identifies the entry-point of the SRv6 End.DT4 behavior processing as reported in Figure 2.2. This function removes the outer IPv6 header along with any other extension header, retrieves the IPv4 packet and invokes the `end_dt_vrf_core()` that performs two important steps:

i retrieving the VRF network interface associated with the specific instance of the SRv6 End.DT4 behavior;

ii invoking the VRF receiving function.

Reference to the VRF network interface, set during the behavior setup phase, is retrieved through the `end_dt_get_vrf()` (step i). Next, such a reference is used by `end_dt_vrf_rcv()` (step ii) to process the IPv4 packet. As a result, the receiving network interface of the processed packet is set to the VRF device associated with the behavior. From now on, the IPv4 packet appears as if it were received directly from the VRF network interface associated with the behavior instance that is processing the traffic. Finally, the route relative to the destination IPv4 address is calculated using the routing table associated with the VRF.

The SRv6 End.DT4 behavior implementation [153] became part of the mainstream Linux kernel, since version 5.11 [91]. The SRv6 End.DT4 behavior is available only if the kernel is compiled with the L3 Master device support (`NET_L3_MASTER_DEV`) enabled.

2.2.6 SRv6 End.DT6 behavior in VRF-mode support in the Linux kernel

SRv6 End.DT6 behavior is defined in the SRv6 network programming [31]. The Linux kernel already offers an implementation of the SRv6 End.DT6 behavior which enables IPv6 L3 VPNs over SRv6 networks. This implementation is not particularly suitable in contexts where operators need to deploy IPv6 L3 VPNs among different tenants which share the same network address schemes. The underlying problem lies in the fact that the current version of SRv6 End.DT6 (called legacy End.DT6 from now on) needs a complex configuration to be applied on routers which requires ad-hoc routes and routing policy rules to ensure the correct isolation of tenants.

Consequently, a new implementation of SRv6 End.DT6 has been introduced with the aim of simplifying the construction of IPv6 L3 VPN services in the multi-tenant environment using SRv6 networks. To accomplish this task, I reused the same VRF infrastructure and SRv6 Endpoint core components already exploited for implementing the SRv6 End.DT4 behavior.

Currently the two SRv6 End.DT6 implementations coexist seamlessly and can be used depending on the context and the user preferences. To support both versions of SRv6 End.DT6, a new attribute (`vrftable`) was introduced to differentiate the implementation of the behavior to be used.

Listing 2.8 shows the `iproute2` command to instantiate a legacy SRv6 End.DT6 behavior making use of the `table` attributes.

Listing 2.8: *`iproute2` command to create a new instance of legacy SRv6 End.DT6 behavior.*

```
$ ip -6 route add 2001:db8::1 encap seg6local action End.DT6 table 100 dev eth0
```

Conversely, Listing 2.9 reports the `iproute2` command to create a new instance of SRv6 End.DT6 in VRF mode where the user is enable to select this new operating mode by using the `vrftable` attribute.

Listing 2.9: *`iproute2` command to create a new instance of SRv6 End.DT6 behavior in VRF mode.*

```
$ ip -6 route add 2001:db8::1 encap seg6local action End.DT6 vrftable 100 dev
    vrf100.
```

Obviously as in the case of SRv6 End.DT4 behavior, the *sysctl* `strict_mode` kernel parameter must be set (`net.vrf.strict_mode=1`) and the VRF associated with the required table must exist.

It is important to note that instances of SRv6 End.DT6 legacy and End.DT6 in VRF mode behaviors can coexist in the same system/configuration without problems.

The VRF mode support for the SRv6 End.DT6 behavior implementation [156] has been merged into the mainstream Linux kernel since version 5.11 [91]. The SRv6 End.DT6 behavior in VRF mode is available only if the kernel is compiled with the L3 Master device support (`NET_L3_MASTER_DEV`) enabled.

2.2.7 SRv6 End.DT46 behavior support in the Linux kernel

SRv6 End.DT46 behavior is defined in the SRv6 network programming [31]. The SRv6 Endpoint subsystem in the Linux kernel only offers SRv6 End.DT4 and End.DT6 behaviors which can be used respectively to support IPv4-in-IPv6 and IPv6-in-IPv6 VPNs. With SRv6 End.DT4 and End.DT6 behavior it is not possible to create a single SRv6 VPN tunnel to carry both IPv4 and IPv6 traffic.

The proposed SRv6 End.DT46 implementation is meant to support the decapsulation of IPv4 and IPv6 traffic coming from a single SRv6 tunnel. The implementation of the SRv6 End.DT46 behavior in the Linux kernel greatly simplifies the setup and operations of SRv6 VPNs.

The SRv6 End.DT46 behavior leverages the infrastructure of SRv6 End.DT4 and End.DT6 behaviors implemented so far, because it makes use of a VRF device in order to force the routing lookup into the associated routing table.

To make the SRv6 End.DT46 behavior work properly, it must be guaranteed that the routing table used for routing lookup operations is bound to one and only one VRF during the tunnel creation. Such constraint has to be enforced by enabling the VRF *sysctl* `strict_mode` parameter as shown in Listing 2.6. Note that the same approach is used for the SRv6 End.DT4 behavior and for the SRv6 End.DT6 behavior in VRF mode.

Listing 2.10 shows the `iproute2` command to create a new instance of SRv6 End.DT46 behavior.

Listing 2.10: *`iproute2` command to create a new instance of SRv6 End.DT46 behavior.*

```
$ ip -6 route add 2001:db8::1 encap seg6local action End.DT46 vrftable 100 dev
    vrf100.
```

The SRv6 End.DT46 behavior implementation [155] has been merged into the mainstream Linux kernel since version 5.14 [93].

2.2.8 Support for SRv6 Micro SID in the Linux kernel

Since release 4.14, the Linux kernel basically offers a subset of the behaviors defined in [31] implementing SRv6 End, End.X, etc. However, the Linux kernel lacks the SRv6 Micro SID (uSID) capabilities and, thus, it does not support any micro segment extension for SRv6 proposed in [28].

In order to provide the support for uSID in the Linux kernel, I designed and implemented a patchset that extends and enhances the existent SRv6 subsystem. The proposed uSID implementation, available at [167], comes up with the support for the uN and uA behaviors which are, respectively, a variant of the Endpoint (End) and of the Endpoint with Cross Connect (End.X). To keep the code clearer, I divided the uN and uA kernel implementations in two separate patches. However, the uA behavior patch must be applied on top of the uN one, since the latter introduce the uSID infrastructure used also for uA.

On the kernel side, both uN and the uA implementations share most of the code. The only difference between the two relies in the way in which the packets are forwarded when no more uSID are available in the IPv6 destination address. Specifically, if a packet contains an SRH header, the uN applies the SRv6 End behavior while the uA applies the End.X behavior. Because the SRv6 uN and uA behaviors are quite similar, in the following I refer only to the uN implementation and I describe the most significant differences with respect to the uA behavior, when needed.

First, I extended the SRv6 infrastructure to support the new structured attribute `struct usid_layout` containing the lengths of the uSID Block and uSID. This enables the user space to create different configurations for uSID Block and uSID for each instance of uN and uA behaviors. Thanks to the support of optional attributes (see Subsection 2.2.1), if the lengths of the uSID Block and/or uSID are not provided at the time of behavior instantiation, the kernel automatically assumes default values for each instance to be created. These values are not fixed in the kernel code, but are read from a new *sysctl* kernel parameter that I created so that they can be changed at runtime as shown in Listing 2.11.

Listing 2.11: *Setting the uBlock and uSID default values for uSID behavior such as SRv6 uN and uA using sysctl kernel parameters.*

```
$ sysctl -w net.ipv6.seg6_local.usid_block_len=32
$ sysctl -w net.ipv6.seg6_local.usid_len=16
```

Besides dealing with the interface between user and kernel space, I also extended the `seg6_local_lwt` structure to store the uSID Block and uSID lengths provided by user space. At the time of creating a new instance, these values are checked to determine whether they are legitimate or not, i.e., values other than 0, uSID Block and uSID lengths multiple of 8 bits, etc. These checks are performed as the behavior instance is created by leveraging the *constructor/destructor callback* patch (see Subsection 2.2.2).

Subsequently, I created two new processing functions called `input_action_un` and `input_action_ua` implementing the processing logic for uN and uA, respectively. These functions are invoked whenever a packet is received and needs to be processed with uN and/or uA. Both the SRv6 uN and the uA behaviors process an incoming IPv6 packet in the same way if the next uSID is not equal to the zeroed *End*

of Carrier (the reserved uSID used to mark the end of a uSID Carrier) and thus they share the same processing logic. In this case, the uN/uA processing logic takes care of updating the IPv6 destination address by consuming the current uSID and replacing it with the next uSID. After that, the uN processing function carries out the route lookup using the updated uSID, while uA on the user-provided uSID adjacency. Based on the result of the operation, the packet could be: discarded or locally delivered or forwarded to the next hop. On the contrary, when the next uSID is equal to the *End of Carrier* (the current uSID is the last one in the IPv6 destination address), uN performs the SRv6 End behavior while uA performs the SRv6 End.X.

In addition, I have also extended the user space `iproute2` suite to support both SRv6 uN and uA behaviors. In particular, using the `ip` command a network operator is able to instantiate and destroy instances of SRv6 uN and uA behaviors. Listing 2.12 shows how simple is to instantiate a new SRv6 uN behavior using the `ip` command as follows:

Listing 2.12: *`iproute2` command to create a new instance of SRv6 uN behavior.*

```
# ip -6 route add bbbb:bbbb:0400::/32 encap seg6local action uN \
    ubl 16 ul 16 dev eth0
```

The new instance of SRv6 uN behavior is set with an uSID Block (`ubl`) of 16 bits and SID (`ul`) of 16 bits. As already introduced before, if the `ubl` and/or `ul` attributes are not provided, the kernel supplies with the default values read by the corresponding *sysctl* kernel parameters. Moreover, both `ubl` and `ul` must assume values which are multiple by 8 bit, otherwise the kernel will abort the operation.

The patched release of `iproute2` for supporting SRv6 uN and uA behaviors is available at [167].

2.2.9 Bugfixes

During my journey through the Linux kernel code, I came across a number of bugs that I later fixed with several patches [99–101] sent to kernel maintainers. It is possible to divide these bugs into two major macro-categories: i) incorrect use of skb; ii) memory leak.

In case (i), I found problems with `skb` header access caused by invalid pointers. An `skb` provides pointers (i.e. `skb->head`) and offsets (i.e. `skb->network_header`), leveraged by the network layers, to keep track of the packet headers inside the `skb` header. There are several helper functions that operate on the `skb` that can change its internal state, including those pointers/header offsets. The `pskb_may_pull()` is an helper function designed to check if there is enough space in the `skb` header to hold a given number of bytes which may invalidates pointers/header offsets. This function has the side-effect of changing the internal state of `skb` and, possibly, invalidating all references to pointers/header offsets. Code that has stored the values of such pointers and uses them after calling a helper function that may affect the `skb` state, could lead to all sorts of problems ranging from kernel panic to memory corruption. The solution to this kind of problem is not complex, since it is enough to update the values of pointers every time an operation that may change the state of the `skb` is performed. However, it is complex and insidious to find out all the accesses to the `skb` header that become

illegitimate because of the side effects of several helper functions (which are very often documented with comments in the code).

In case (ii), I identified memory leaks caused by `skb`s that were not properly released when packet processing failed, for example, in `seg6_input_core()` function. I caught the leaks by digging into the code and thanks to `kmemleak` [102] tool I was able to confirm these issues. `kmemleak` provides a way of detecting possible kernel memory leaks in a way similar to a tracing garbage collector. Then, I fixed the code which I tested once again using this tool and I found that those memory leaks were gone.

2.3 The extended Berkeley Packet Filter (eBPF)

In this Section, I introduce the extended Berkeley Packet Filter (eBPF) and, without any claim to be exhaustive, I outline its main and most interesting aspects. eBPF is a (kernel) technology that enables programs to be run in the kernel safely, without the need to insert modules or modify kernel sources. In the following Subsections, I discuss fundamental aspects that are useful for understanding how I built, on eBPF, network programming and traffic processing solutions, which will be presented in following Chapters. A minimal understanding of eBPF is, therefore, necessary to highlight the disadvantages and advantages of this technology, especially when compared with classical network programming approaches based on kernel modules and user space applications.

2.3.1 The extended Berkeley Packet Filter (eBPF)

Proposed in the early '90s, the Berkeley Packet Filter (BPF) [108] is designed as a solution for performing packet filtering directly in the kernel of BSD systems. BPF comes along 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. The bytecode of an BPF application is transferred from the user space to the kernel space where it is checked to ensure security and prevent kernel crashes. BPF also defines a packet-based memory model, certain registers and a temporary auxiliary memory.

The Linux kernel has supported BPF since version 2.5 and the major changes over the years have been focused on implementing dynamic translation [75] between BPF and x86 instructions. Starting from release 3.18 of the Linux kernel, the extended BPF (eBPF) [50] represents an enhancement over BPF which adds more resources such as new registers, enhanced load/store instructions and it improves both the processing and the memory models. eBPF programs can be written using assembly instructions later converted in bytecode or in restricted C and compiled using the Clang/LLVM compiler [128]. The bytecode has to be loaded into the system through the `bpf()` syscall that forces the program to pass a set of sanity/safety checks performed by the *verifier* integrated into the Linux kernel. In fact, eBPF programs are considered untrusted kernel extensions and only safe eBPF programs (i.e. the ones that can not be harmful for the system)[6] can be loaded successfully in the system. Conversely, the eBPF programs

[6]The verification step assures that the program cannot crash, that no information can be leaked from the kernel to the user space, and it always terminates.

considered unsafe are rejected. To be considered safe, a program must satisfy a number of constraints such as limited number of instructions, limited use of backward jumps, limited use of the stack, etc[7]. These limitations [111] can impact on the ability to create powerful network programs.

2.3.2 Events and Hooks

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. Within the Linux kernel there are several *hooks* to which eBPF programs are attached. Each of these hooks is related to a specific type of event, i.e. receiving a network packet. When a specific event occurs, the kernel retrieves the related hook and executes the attached eBPF program. Since the hooks are different from each other, the execution contexts of the eBPF programs also differ. Thus, according to the context, there are programs that can execute network functions as soon as packets hit the NICs or can legitimately carry out socket filtering operations, while others can perform traffic classification at the TC layer and so on.

2.3.3 Maps

eBPF programs are designed to be stateless so that every run is independent from the others. To this regard, the eBPF infrastructure provides specific data structures, called *maps*, that can be accessed by the eBPF programs and by the user space when they need to persist and/or share some information.

Map types include hash tables or arrays, ring buffer, stack trace, least-recently used, longest prefix match, and more.

2.3.4 Helper functions

An eBPF program cannot arbitrarily call into a kernel function mainly for two reasons: i) security and safety; ii) avoid being bound to a specific version of the kernel. Helper functions are APIs provided by the kernel which allow programs to interact with the system or with the context where they work. For instance, they can be used to print debugging messages, to get the time since the system was booted, to interact with eBPF maps, or to manipulate network packets. Since there are several eBPF program types, and that they do not run in the same context, each program type can only call a subset of those helpers.

2.3.5 Tail call and function call

A fundamental concept in eBPF is the *tail call*. Tail calls could be considered as a mechanism in eBPF which allows one eBPF program to call another eBPF program, but without returning back to the *caller* program. This approach comes with a minimal overhead as unlike the traditional *function call* approach, since it is implemented by transferring the execution control to the `callee` and reusing the `caller` stack frame.

Tail calls allow to concatenate eBPF programs with each other, whereby every program can be verified independently one from another. Using the tail call approach

[7] An eBPF program is non-Turing complete by design.

however makes life harder for developers since it does not allow directly to pass any context between the *caller* and *callee* eBPF programs.

There are two components involved for executing a tail call: i) a special eBPF maps, populated by the user space, with *key/values* where *keys* are IDs (numbers) chosen by the user and *values* are the file descriptors of the tail called eBPF programs; ii) a helper function `bpf_tail_call()` where the arguments are the context, a reference to the special map discussed above, and the ID assigned to the eBPF program to be tail called.

A more recent feature was introduced into the eBPF core which enables eBPF to eBPF calls. Before that, eBPF C programs were forced to declare any reusable code (i.e. functions) as *inline* so that the LLVM could compile and generate the ELF object file. Consequently, all the defined functions were *inlined* and, therefore, duplicated many times in the resulting file object, artificially inflating the code size. For these reasons, eBPF designers supported the eBPF to eBPF call which no longer needs the inlining of the *callee* functions. Such improvements, however, come with some restrictions. First of all, mixing tail calls and eBPF to eBPF calls were not possible on kernel releases before 5.10. Starting from that release, they could be mixed but only on x86-64 bit architecture. Moreover, when the verifier detects a mixed use of these two calling approaches, the eBPF program stack is limited in size for avoiding possible overflows.

2.3.6 The eXpress Data Path (XDP)

The eXpress Data Path (XDP) proposed in [70] 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 "traditional"[8] 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 ([169], [144], [174], [171]).

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

it is necessary to mention that eBPF programs hooked to XDP cannot be used to process traffic that is either generated locally or is about to be transmitted over the network through the NIC. Other hooks exist within the kernel to process this type of traffic, such as the ones built into Linux Traffic Control (TC).

[8] By the term "traditional" I refer to the Linux kernel networking stack that does not take advantage of any eBPF virtual machine-based packet processing capability.

2.4 State-of-the-art: alternatives to the SRv6 Linux kernel subsystem

2.4.1 SRv6 support with SREXT module

The SREXT module [3] is an implementation of the SRv6 network programming model. When it was designed, the Linux kernel only offered the basic SRv6 processing (SRv6 End behavior). SREXT is an external kernel module that complemented the SRv6 Linux kernel implementation providing a set of behaviors that were not yet supported. Currently most of the behaviors implemented in SREXT are supported by the mainline Linux kernel (with the exception of the SR-proxy behaviors which will be discussed in Chapter 4). SREXT adopts the concept of *My Local SID* realized through table *Localsid* that contains the local SRv6 segments instantiated explicitly in the node and associates each SID with a function. Unlike the SRv6 Linux kernel implementation, the *Localsid* table is completely independent from the Linux routing subsystem and is meant to contain only SRv6 SIDs. Hence, each entry of the *Localsid* table is an SRv6 segment that is associated with an SRv6 endpoint behavior.

Once the SREXT module is loaded, it registers a callback function in the pre-routing hook of the netfilter [119] framework. Such callback is invoked each time an IPv6 packet is received and processed by the `PREROUTING` stage (see Figure 2.1). If the destination address of an IPv6 packet matches an entry of the *Localsid* table, then the correspondent behavior is applied; otherwise the processing continues in the traditional IPv6 receiving path, eventually reaching out the routing decision.

However, SREXT exhibits a number of limitations. First of all, SREXT does not support network namespaces and this limits its deployment in the presence of system virtualization systems such as Linux Containers, Dockers, LXD, etc. SREXT intercepts network traffic in the `PREROUTING` processing stage, basically "stealing" it from the IPv6 receiving path of the kernel. Although this may allow in some cases an increase in SRv6 traffic processing performance, on the other hand it prevents the direct access to some important features such as Virtual Routing and Forwarding (VRF). SREXT is developed as a kernel module, and because of the way the Linux kernel is designed, it does not have access to all internal kernel capabilities unless they are properly designed to be accessed by a module. This fact leads to code duplication in SREXT that needs to be reimplemented, i.e. decap packet helper functions present in the Linux SRv6 kernel subsystem are not available, at the moment, to be accessed externally by a module. In addition, SREXT is provided as an out-of-the-tree kernel module and this requires that it must always be updated with the latest kernel versions and, at the same time, compatible with previous ones.

From a configuration point of view, behaviors cannot be configured in SREXT using the well-known `iproute2` tool-suite. The lack of integration of SREXT with the SRv6 subsystem of the Linux kernel forces the adoption of a custom user-space configuration tool.

2.4.2 SRv6 support in Virtual Packet (VPP)

During the last decade, the concept of implementing user space software packet forwarders gained a lot of attention from both academia and industries. Despite the in-kernel packet processing approach, the core idea underneath packet processing in

user space consists in mapping the network interface card (NIC) directly into the user land bypassing the kernel. Bypassing techniques rely on a sort of direct memory access (DMA) ring buffers [139] which are shared between the NIC driver and the user space. The NIC driver retrieves on-the-wire packets and stores (produces) them in a ring buffer where an user space forwarder application is able to consume those packets. Conversely, when the forwarder needs to send packets, it stores them in the ring buffer and the NIC driver takes care of sending such packets on the wire as soon as possible. Netmap [136] and the Data Plane Development Kit (DPDK) [42] are two famous examples of kernel bypassing mechanisms. The latter is the most widely deployed and supported by several NICs from various vendors [43]. In fact, to bypass the network stack of the Linux kernel, the NIC must use a special driver that excludes the kernel from processing packets in favor of storing them in the shared buffer accessed by the forwarder in user space. DPDK enables very high packet processing rates since it offers major efficiency with respect to the kernel networking stack. Indeed, performance is boosted basically due to the fact that DPDK tries to avoid copying the same packet data multiple times as well as it operates in batches of packets to be cache efficient. It is worth noting that DPDK only provides a subset of features (i.e.: IPv4 and IPv6 over Ethernet, but not ATM) that the Linux kernel does, so the code is more specialized on a specific task. Several frameworks for fast packet processing are build on top of DPDK, i.e. Virtual Packet Processing (VPP) [176] and Network Function Framework for GO (NFF-GO) [120]. VPP is an open source virtual switch and router which provides high performance forwarder capabilities run on commodity hardware. VPP is designed to be flexible and modular so that new functionality can be added through the introduction of new *plugis*. Packet processing in VPP consists of graph nodes that are connected together. Each graph node performs one function of the processing stack such as IPv4 packet input or IPv6 FIB look-up. The way in which graphs are composed together is not written in the stone; it can be configured at runtime, indeed. VPP is built on top of DPDK and it supports packet processing in batch mode [36]. This means that each graph node could be seen as a processing stage where: i) it receives a batch of packets from the preceding stage; ii) it applies the specific processing function to that batch of packets; iii) it passes the control to the next processing stage. Batch technique (implemented also in the Linux kernel) aims to improve performance by leveraging the CPU instruction cache. VPP enables very high packet processing speeds that are an order of magnitude higher than those achievable using Linux kernel networking. VPP supports SRv6 capabilities since the release 17.04. Most of the SRv6 Endpoints behavior defined in [27] are supported (i.e.: End, End.X, End.T, etc). Hence, new graph nodes have been added in VPP to support SRv6 packet processing. An API has been added to allow developers to create, in a simplified way, new SRv6 Endpoint behaviors using the VPP plugin framework.

However, to take full advantage of the VPP capabilities, network cards must support DPDK. However, this is not always guaranteed especially in embedded systems using OpenWRT or LEDE based distributions. A user space packet processing framework based on kernel bypassing can not directly exploit any network services made available by the kernel. This implies that such frameworks must necessarily re-implement all protocols (where possible) needed to address specific use cases. Moreover, configuration tools such as `iproute2` or `iptables` can not be used to configure VPP as

they clearly act on the kernel networking stack. In other words, VPP on DPDK literally replaces the Linux kernel networking and, for this reason, it is not possible to have an integrated solution with the services already offered by the kernel itself. It is worth noting that the CPU utilization changes between the packet processing done by VPP/DPDK and the Linux kernel. In the Linux kernel networking case, the processing of packets is interrupt-oriented (mitigated by NAPI to increase performance). This implies that only in the event of network traffic the kernel spends CPU cycles to process packets. On the contrary, in VPP it is necessary to reserve a number of processors that in polling mode constantly check for network packets stored inside the circular buffer shared between the NIC driver and the user space forwarder application. Regardless of the received traffic, the CPUs dedicated to this task are constantly active and this is a waste of valuable computing resources. It is also necessary to adapt the Linux scheduling policies so that these polling tasks are executed on CPUs in isolated mode. This is necessary when there is the need to reduce traffic processing latency due to context changes between other processes and the forwarder. A traffic processing model based on constant polling mode is not optimal in virtualized environments, where computing resources are assigned dynamically and consumption of resources must be minimized to achieve server consolidation. One must also consider the energy issues due to the fact that there are CPUs consuming electricity when actually there are no packets to process in the system. This obviously also affects the cooling performance of the system and therefore the costs associated with air conditioning.

CHAPTER 3

A performance evaluation framework for SRv6 based networks

The work I present in this Chapter is mostly taken from my paper *SRPerf: a Performance Evaluation Framework for IPv6 Segment Routing*, published at *IEEE Transaction on Network and Service Management*, 2020 [133].

3.1 Introduction

Nowadays, Segment Routing over IPv6 (SRv6) is supported on several hardware routers produced by different vendors (i.e.: Cisco, Juniper) as well as on software ones based on the Linux kernel and the Vector Packet Processing [176]. Since SRv6 can be chosen as an enabling technology for both operator networks and data centers, it is crucial to investigate non-functional characteristics such as performance, scalability and fault tolerance. In this regard, I have contributed to the design and development of a new framework named SRPerf [133], an open source platform for Segment Routing performance evaluation. In addition to being a viable tool for evaluating performance, the SRPerf framework must also be a reasonably easy-to-use tool, extend, and run on commodity hardware. Designing and implementing something similar to SRPerf is a non-trivial task because, on general purpose systems, it is not easy to generate and process traffic at high speed considering the limited time frame/budget for a CPU to operate on a single network packet. SRPerf is designed following the guidelines provided by IETF regarding performance evaluation of forwarding devices [20]. Such guidelines provide valuable indications on how the test network should be set up, the packet format, and the different measurements that can be collected on the forwarding devices.

In this Chapter, I present the SRPerf framework used both to test the performance of the Linux SRv6 kernel implementation and to quantitatively assess the impact of my proposed solutions and extensions to the networking stack discussed in this thesis work. The ultimate goal of my research work is to strive for efficient solutions and, for this reason, the feedback gathered from a tool like SRPerf is invaluable.

The Chapter is structured as follows. In Section 3.2, I briefly discuss the basic principles adopted for designing a performance evaluation and testing framework. Then, in Section 3.3, I describe the design of SRPerf by highlighting its main features. All experiments presented in this thesis are carried out over the same testbed setup which is described in Section 3.4. Note that SRPerf has been designed to support several data planes in addition to the one provided by Linux kernel networking. Regarding that, an extension for SRPerf has been developed enabling the framework to interact with Linux-based software routers whose data plane is implemented through VPP [176]. The description and results of performance tests related to SRv6 Endpoint counters, SRv6 End.DT4, End.DT6 (VRF mode), End.DT46 behaviors and Micro SID are reported in Section 3.5. In Section 3.6, I investigate the state-of-the-art of existing techniques and performance frameworks designed to evaluate SRv6 performance. Finally, I draw some conclusions in Section 3.7.

3.2 Design a performance evaluation and testing framework

IETF defined in RFC 2544 [20] a set of guidelines for network benchmarking that could be summarized as follows:

- *Testbed setup and benchmarking measures*: a tester node equipped with two NICs is required for benchmarking a *Forwarder*. As shown in Figure 3.1, the tester node sends generated traffic to the *Forwarder* through one NIC, while it receives back the processed traffic on the other one. In this way, the tester is in charge of evaluating different types of performance measurements defined by the IETF such as delay, latency and throughput;

- *Device configuration*: the System Under Test (SUT) must be configured to process the received traffic according to the function to be tested and, then, forward it back to the tester;

- *Frame size*: performance of the *Forwarder* device can depend on the packet size. For this reason, RFC 2544 provides a list of frame sizes to be considered during tests;

- *Line rate*: the maximum number of frames per second that can be sent from the tester towards the *Forwarding* device is defined as Line rate. It is calculated by dividing the link maximum speed by the test frame size, including also the protocol overhead;

- *Trial description and duration*: several repetitions of the same test are required for carrying out reliable performance experiments. Hence, each trial should be always defined carefully to guarantee consistency among the repetitions for obtaining accurate measurements. In addition, the duration of each repetition has to

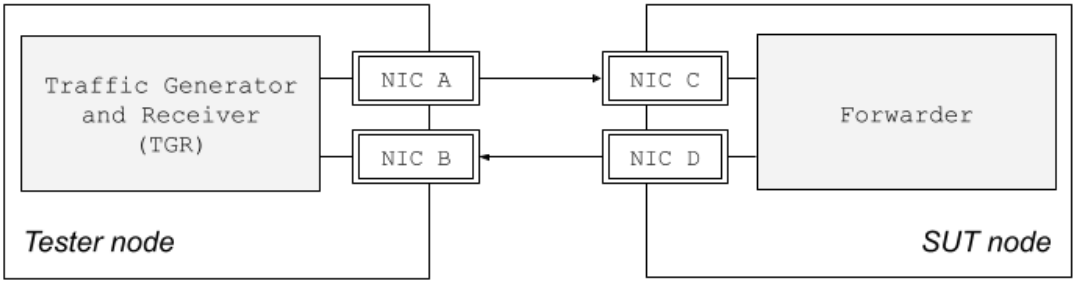


Figure 3.1: *Testbed setup for performance evaluation experiments.*

consider the specific forwarding function to be tested in order to avoid abnormal results.

3.3 SRPerf framework

In this section, I illustrate the SRPerf framework. In particular, I describe the internal design and the high-level architecture in Subsection 3.3.1. In Subsection 3.3.2, I introduce the the evaluating methodology as well as I describe the *Partial Drop Rate* (PDR) metric used to characterize the performance of the forwarding node. Finally in Subsection 3.3.3, I explain the algorithm that I developed and implemented in SRPerf for finding the PDR of a given forwarding function.

3.3.1 SRPerf Design and Architecture

SRPerf has been designed following the network benchmarking guidelines defined in RFC [20] by IETF. From an architectural point of view, SRPerf is composed of two main functional blocks: the *Testbed* and the *Orchestrator*. Testbed comprises the Traffic Generator and Receiver (TGR) and the System Under Test (SUT). The two components (also referred to as *nodes*), which may be complex systems, are connected back-to-back using two NICs as illustrated in Figure 3.1. TGR generates and sends traffic on NIC `A` which is received by NIC `C` of the SUT. Then, the SUT processes the traffic which is sent back to the TGR through NIC `D`. Finally, TGR receives traffic on NIC `B` and, at this point, it collects different performance measurements such as throughput and round-trip delay.

TGR relies on the TRex [161] for generating traffic. TRex is an open source generator based on the Data Plane Development Kit (DPKD) [42] for fast packet processing on commodity hardware. TRex is able to generate virtually any type of traffic reaching up to 10-22 millions of packets per second (mpps) on a single CPU. It provides per-interface statistics, handled at hardware layer, which are used to collect performance measurements such as throughput and delay. TRex offers a rich Python API that can be used by third-party (like SRPerf) to manage traffic generation and packet manipulation.

With regard to the SUT node, SRPerf natively supports the Linux kernel networking stack as *Forwarder*. It is also possible to support VPP which replaces the kernel networking stack by moving packet processing in user space. SRPerf has been extended to interact with the VPP data plane since de-facto network tools such as `iproute2` can not interact with the VPP networking stack.

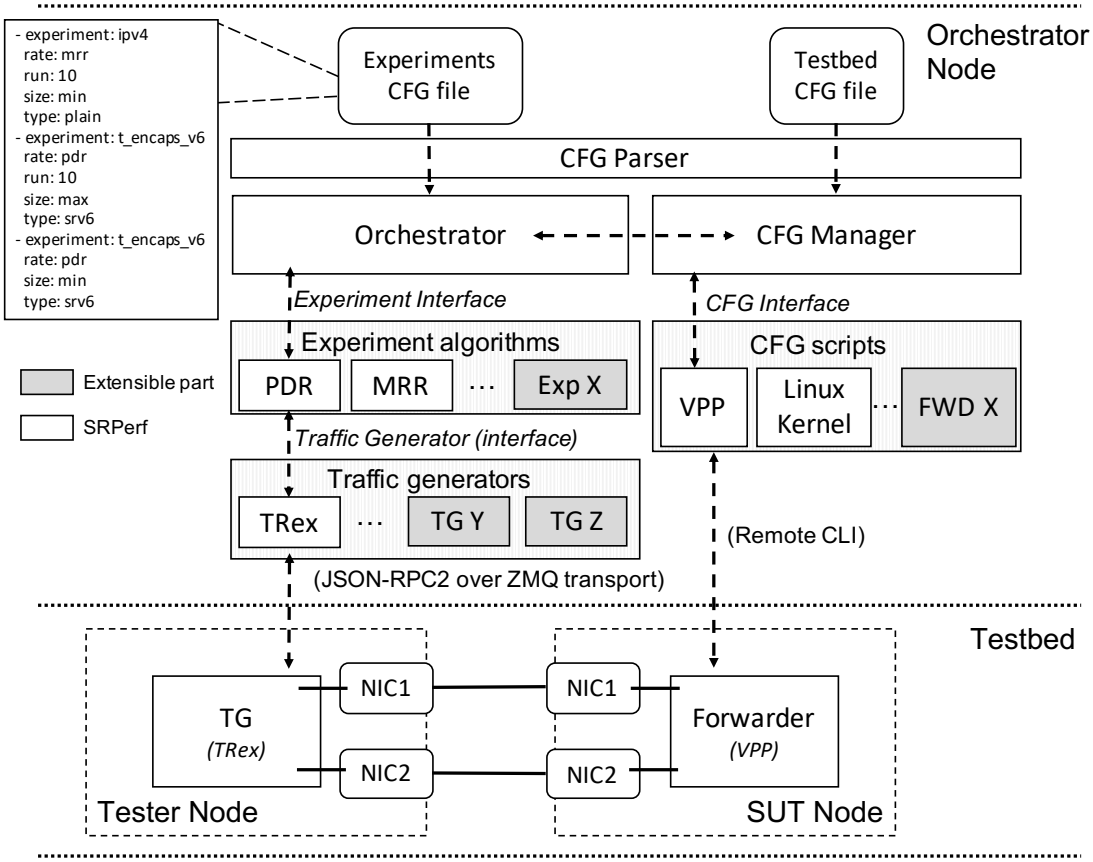


Figure 3.2: *Overall architecture of the SRPerf framework.*

I follow a top-down approach to illustrate the SRPerf architecture depicted in Figure 3.2. First of all, the SRPerf is made of two main software components which are the *Orchestrator* and the *CFG Manager*. The Orchestrator is in charge of setting up and deploying the experiments described in the *Experiments CFG* yaml file which defines: i) the type of experiment, i.e. the forwarding function to be tested, the type of algorithm for evaluating performance; ii) the number of runs; iii) the size and type of packet to be exchanged, during the experiment, between the TGR and the SUT. The Orchestrator makes also use of another configuration file called *Testbed CFG* needed to log into the SUT and to enforce, in the Forwarder, the configuration (i.e.: setting up routes, network addresses, etc) required by the experiment that is going to be run.

The Orchestrator automates the provisioning and the execution of each experiment (as described in the configuration file) that I modeled in the SRPerf framework through the *Experiment* interface. Specifically, the Orchestrator can use different algorithms for calculating the throughput. Currently, I designed two different algorithms and both are supported by SRPerf: the MRR (Maximum Receive Rate) and the PDR (Partial Drop Rate) explained in Subsection 3.3.3. Moreover, the Orchestrator provides a mapping between the forwarding behaviors to be tested and the type of traffic required to test each behavior.

I designed and implemented a high level abstraction layer used by the Orchestrator to control the TGR. The purpose of this layer consists of translating the requests, (i.e.: start/stop sending traffic, read packet counters, etc) coming from SRPerf components, in commands to be executed on the TGR without exposing low level details of the TRex API. Specifically, each driver is a Python wrapper that can speak native Python APIs or use any other transport mechanism supported by the language. For example, the TRex driver includes the Python client of the TRex automation API [162] that uses as transport mechanism JSON-RPC2 [77] over ZMQ [187]. For these reasons, the Orchestrator can be deployed on the same node of the TGR or in a remote one.

The CFG Manager controls the forwarding engine in the SUT. It is responsible for enforcing the required configuration in the Forwarder. The Orchestrator provides the mapping between the forwarding behaviors to be tested and the required configuration of a given forwarding engine. For each forwarding engine, SRPerf implements a *CFG script* which provides the CFG Manager with the means to enforce a required configuration. In particular, a CFG script is a bash script defining a configuration procedure for each behavior to be tested. For example, in the CFG Script for the Linux kernel I implemented the procedure for testing SRv6 End.DT4 behavior in VRF-mode, the SRv6 End.DT6 behavior in VRF-mode and SRv6 counters.

The configuration is applied using the Command Line Interface (CLI) exposed by the forwarder. The CFG Manager first pushes the CFG scripts in the SUT and then enforces a given configuration, by running commands over an SSH connection.

3.3.2 Evaluation methodology

RFC 1242 [21] and RFC 2544 [20] define the device *throughput* as the maximum rate at which all received packets are forwarded by the device. Throughput can be expressed in number of bits per second (bps) as well as number of packets per second (pps). It represents a standard metric for comparing performance of networking devices. FD.io CSIT Report [35] defines *No-Drop Rate* (NDR) and *Partial Drop Rate* (PDR) where:

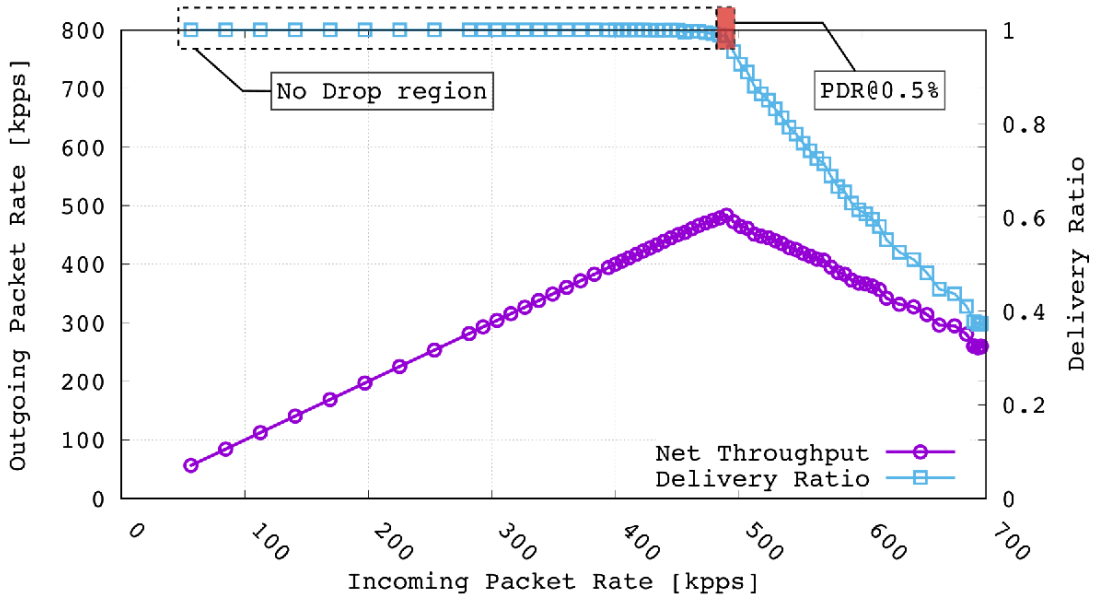


Figure 3.3: *Throughput, Delivery Ratio (DR) and Partial Drop Rate (PDR).*

- NDR is the highest forwarding rate achieved without dropping packets and then it corresponds to the throughput defined by [21] and [20];

- PDR is the highest receiving rate supported without traffic loss exceeding a given pre-defined loss ratio threshold.

In the following, I use the notation PDR@X% where X represents the loss ratio threshold. For example, a PDR@0.5% (where X = 0.5) means that the achieved throughput on a given forwarding device has a drop rate of 0.5%, at most. Since the throughput term can be used with wider meanings, the NDR/PDR terminology is clearer and less ambiguous and will be used hereafter. As a consequence, I will use throughput to refer in general to the output forwarding rate of a device. Moreover, the PDR concept is more generic than NDR and, for this reason, in this thesis I will only consider the former.

Characterizing the PDR requires the scanning of a broad range of possible traffic rates. To explain the process, I consider the plain IPv6 forwarding in the Linux kernel. Figure 3.3 plots the throughput (i.e. the output forwarding rate) and the *Delivery Ratio* (DR) versus the input rate, defined and evaluated as follows. I generate traffic at a given packet rate PS [kpps] for a duration D [s] (usually $D = 10s$ in chosen as the default values in these experiments). Let P_{IN} (Packets INcoming in the SUT) be the number of packets generated by the TGR node and incoming to the SUT in an interval of duration D. I define the number of packets transmitted by the SUT (and received by the TG) as P_{OUT} (Packets OUTgoing from the SUT). The throughput T is P_{OUT}/D [kpps]. Therefore, the DR is defined as follows:

$$P_{OUT}/P_{IN} = P_{OUT}/(PS * D) = T/PS$$

The DR can be considered as the ratio between the input[1] and the output packet rates of a device, for a given forwarding function (i.e. an SRv6 behavior) under test.

In Figure 3.3, starting from the left (low PS) there is a region in which the throughput increases linearly with the PS and the Delivery Ratio is 1. Ideally, the Delivery Ratio should remain 1 (i.e. no packet loss) until the SUT saturates its resources and starts dropping a fraction of the incoming traffic. This is defined as the *No Drop* region and the highest incoming rate for which the Delivery Ratio is 1 is defined as *No Drop Rate* (NDR). In all experiments where the Forwarder is based on the Linux kernel networking, a very small but not negligible packet loss ratio is measured in a region in which there is a (nearly) linear increase in throughput. Therefore, according to [35] I define a Partial Drop Rate (PDR) by setting a threshold for the measured loss ratio, typically I used 0.5% as threshold corresponding to a Delivery Ratio of 0.995. The PDR@0.5% is the highest rate at which the Delivery Ratio is at least 0.995.

The value of the PDR lies in the fact that it enables the characterization of a given SUT configuration with a single scalar value, rather than considering the complex relation between throughput and incoming packet rate as shown in Figure 3.3. When the incoming packet rate overcomes the PDR, the throughput starts to decrease due to trashing phenomena.

3.3.3 Partial Drop Rate (PDR) algorithm

Algorithm 1 PDR finder algorithm

Input: $startingTxRate, lineRate, \epsilon, pdrThreshold$
Output: $minRate, maxRate$

$minRate \leftarrow startingTxRate$
$maxRate \leftarrow lineRate$
while $|maxRate - minRate| > \epsilon$ **do**
    $txRate \leftarrow \frac{maxRate + minRate}{2}$
    $rxRate \leftarrow runExperiment(txRate)$
    $deliveryRatio \leftarrow \frac{rxRate}{txRate}$
    if $deliveryRatio < pdrThreshold$ **then**
        $maxRate \leftarrow txRate$
    else
        $minRate \leftarrow txRate$
    end if
end while

Estimating the Partial Drop Rate (PDR) for a given forwarding function (i.e. an SRv6 behavior) is both time consuming and computationally intensive since a large set of possible traffic rates must be investigated. In order to automate the process of finding the PDR, I designed and implemented the *PDR finder algorithm*. Starting with a given range of traffic rates to be tested, the algorithm scans that range to estimate the value of the PDR. The algorithm guarantees that the error (ϵ) on the PDR found is limited and therefore lower than a maximum value, fixed a priori.

[1] For each incoming packet rate, I run a number of test repetitions (i.e. 15) to evaluate the average and the standard deviation of the outgoing rate.

In Algorithm 1, I report the pseudo-code of the PDR finder algorithm. It leverages the binary search approach to determine the approximate value of the PDR within the solution space limited above by the NIC transmitting line rate. On termination, the algorithm returns the interval $[minRate, maxRate]$ where $|maxRate - minRate| \leq \epsilon$ and ϵ corresponds to the confidence of the PDR value. In other words, the real (unknown) value of the PDR lies in the range $[minRate, maxRate]$ and the error between the real PDR value and the estimated one is less than or equal to ϵ.

Initially, the algorithm sets the searching interval to $[minRate, maxRate]$ where $minRate = startingTxRate$ and $maxRate = lineRate$. The $lineRate$ is the maximum traffic rate that the NIC can process, while the $startingTxRate$ is the minimum traffic rate chosen[2] to avoid any loss in the Forwarder. After the setup phase, the algorithm enters the loop if the condition (line 3) is verified. In particular, the condition checks whether the searching interval width is still larger than the error (ϵ) between the real (unknown) value of the PDR and the estimated value. In case the condition is not satisfied, the loop ends and the algorithm returns the searching interval $[minRate, maxRate]$ used for deriving the approximated PDR value. Otherwise, the algorithm enters the body of the loop for calculating the average value ($txRate$) of the searching interval (line 4). This value is used for sending traffic, at the given rate, to the Forwarder during the experiment (line 5). On experiment completion, the rate of the traffic leaving the Forwarder is used for determining the Delivery Ratio (line 6). The DR is compared to the threshold value ($pdrThreshold$) fixed on the basis of the PDR%X. Based on the comparison outcome, the searching interval is halved properly (lines 7-11) as follows. When the DR is less than $pdrThreshold$, it means that the Forwarder is not able to process the traffic at the same rate as it arrives. Consequently, the algorithm halves the size of the searching interval by updating the upper bound with $txRate$ (line 8). Conversely, when the DR is greater than or equal to $pdrThreshold$, it means that the Forwarder is able to process the incoming traffic with zero or less than X% loss. Therefore, the size of the searching interval is halved by updating the lower bound with $txRate$ (line 10).

3.4 Testbed

I used Cloudlab [33] for deploying and running any performance test on a testbed organized as illustrated in Figure 3.1. CloudLab is a cloud infrastructure dedicated to scientific research on the future of cloud computing. The testbed is composed of two identical nodes (Tester and SUT). Each of these nodes is a bare metal server whose specs are summarized in Table 3.1. On both Tester and SUT machines, I deployed the SRPerf framework used for carrying out the performance tests. On the SUT machine, I installed different Linux kernel versions depending on the features that I had to test. Along with that, I also had to keep aligned the user space tools such as `iproute2` and `ethool` [51] versions with the kernel one. I used the `iproute2` tool-suite for controlling the Linux kernel data plane (i.e. configuring SRv6 behavior and the forwarding subsystems), while `ethool` for acting directly on NIC(s) hardware features. In this regard, I disabled hardware offloading capabilities offloading such as large receive offload (LRO) [34], generic receive offload (GRO) [96], generic segmentation

[2]For instance, it could be chosen the trivial of 1 pps as the minimum traffic rate.

offload (GSO) [185] and checksum offloading. Along with that, I also turned off the Hyper-Threading CPU capability since it can impact on the forwarding performance as indicated in [146], [159]. I completed the SUT configuration by forcing all the received/sent traffic to be processed by a single CPU core. To accomplish this task, I had to configure the receive-side scaling (RSS) [97] and the IRQ affinity [98] to assign all the hardware-based receiving queues to the same CPU core. This configuration aims to measure the Linux kernel performance in packet processing (on a single CPU core) independently of the offloading capabilities offered by the modern NICs.

Table 3.1: *TGR/SUT hardware specs.*

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

3.5 Performance evaluation of SRv6 behaviors in the Linux kernel

The SRPerf framework has been used for measuring the performance of SRv6 behaviors implemented in the Linux kernel. Tests have been classified in different categories on the basis of the operations carried out on packets, i.e. Plain IPv6 forwarding, transit behaviors, endpoint behaviors w/o decapsulation and so on. A detailed descriptions of such tests along with results can be found in [133]. I leveraged the SRPerf framework also for testing the performance of the extensions and new SRv6 capabilities, described in Chapter 2, that I have introduced in the Linux kernel mainline. In particular, in the following Subsections I report the performance for the SRv6 Behavior counters, the new SRv6 End.DT4, End.DT6 (VRF mode) and End.DT46 behaviors.

3.5.1 Impact of counters for SRv6 Endpoint Behaviors on performance

To determine the performance impact due to the introduction of counters in the SRv6 Endpoint behavior subsystem, I have carried out extensive tests. I chose to test the throughput achieved by the SRv6 End.DX2 Behavior because, among all the other behaviors implemented so far, it reaches the highest throughput which is around 1.5 Mpps (per core at 2.4 GHz on a Xeon(R) CPU E5-2630 v3) on kernel 5.12-rc2 using packets of size $\tilde{1}00$ bytes. Three different tests were conducted in order to evaluate the overall throughput of the SRv6 End.DX2 behavior in the following scenarios:

1. vanilla kernel (without the SRv6 Endpoint behavior counters patch) and a single instance of an SRv6 End.DX2 behavior;
2. patched kernel with SRv6 Endpoint behavior counters and a single instance of an SRv6 End.DX2 behavior with counters turned off;
3. patched kernel with SRv6 Endpoint behavior counters and a single instance of SRv6 End.DX2 behavior with counters turned on.

All tests were deployed and carried out using the testbed described in this Chapter. Results of tests are shown in Table 3.2.

Table 3.2: *SRv6 Endpoint behavior counters performance.*

Scenario	Throughput (avg.)	Std. Dev.
1	1504.76 kpps	3.96 kpps
2	1501.47 kpps	2.98 kpps
3	1501.32 kpps	2.96 kpps

As can be observed, throughputs achieved in scenarios (2),(3) did not suffer any observable degradation compared to scenario (1).

3.5.2 SRv6 End.DT4, End.DT6 (VRF mode) and End.DT46 performance

I provided the support for SRv6 End.DT4 and End.DT6 (VRF mode) behaviors in the mainline Linux kernel. Once I implemented the needed infrastructure for implementing such behaviors, I came up with the SRv6 End.DT46 behavior patch. All design and implementation phases were guided by the goal of avoiding code duplication and maximizing as much as possible the infrastructure already in place for SRv6 End.DT4, End.DT6 behaviors. At the same time, I wanted to keep the processing overhead very low. Therefore, in order to verify my solutions, I tested the performance of SRv6 End.DT46 Behavior and compared it with the performance of SRv6 End.DT4 and End.DT6 behaviors, considering both the patched kernel and the kernel before applying the SRv6 End.DT46 behavior patch (referred to as vanilla kernel). In details, the following decapsulation scenarios were considered:

1.a IPv6 traffic in SRv6 End.DT46 behavior on patched kernel;

1.b IPv4 traffic in SRv6 End.DT46 behavior on patched kernel;

2.a SRv6 End.DT6 behavior (VRF mode) on patched kernel;

2.b SRv6 End.DT4 behavior on patched kernel;

3.a SRv6 End.DT6 behavior (VRF mode) on vanilla kernel (without the End.DT46 patch);

3.b SRv6 End.DT4 behavior on vanilla kernel (without the End.DT46 patch).

All tests were deployed and carried out using the testbed described in this Chapter. I considered IPv4/IPv6 traffic handled by a single core (at 2.4 GHz on a Xeon(R) CPU E5-2630 v3) on kernel 5.13-rc1 using packets of size $\tilde{1}00$ bytes and results are shown in Table 3.3.

Considering the results for the patched kernel (1.a, 1.b, 2.a, 2.b), is it possible to observe that the performance degradation incurred in using the SRv6 End.DT46 behavior rather than SRv6 End.DT6 and SRv6 End.DT4 behaviors respectively for IPv6 and IPv4 traffic is minimal, around 0.9% and 1.5%. Such very minimal performance degradation is the price to be paid if one prefers to use a single tunnel capable of handling both types of traffic (IPv4 and IPv6).

Comparing the results for the SRv6 End.DT4 and the End.DT6 behaviors under the patched and the vanilla kernel (2.a, 2.b, 3.a, 3.b), I observe that the introduction of the

Table 3.3: *SRv6 End.DT4, End.DT6 (VRF mode) and End.DT46 behaviors performance.*

Scenario	Throughput (avg.)	Std. Dev.
1.a	684.70 kpps	0.70 kpps
1.b	711.69 kpps	1.20 kpps
2.a	690.70 kpps	1.21 kpps
2.b	722.22 kpps	1.72 kpps
3.a	690.02 kpps	2.60 kpps
3.b	721.91 kpps	1.22 kpps

SRv6 End.DT46 behavior patch has no impact on the performance of SRv6 End.DT4 and SRv6 End.DT6 behaviors.

3.5.3 Performance assessment for Micro SID

To determine the quality of the Micro SID implementation and in particular of SRv6 uN Behavior, I conducted various experiments using SRPerf and the Cloudlab environment as described in this Chapter. Specifically, on the SUT I installed a patched Linux kernel version 5.6 that includes my implementation of the SRv6 uN behavior. Then, I considered five different experiments, each with a specific combination of SRv6/uN function and packet type:

1. function `uN(un)` with IPv6-in-IPv6 encapsulation without SRH. In this experiment the Micro SIDs are encoded directly within the destination address of the IPv6 header and the packets processed are the smallest ones of this measurement campaign (118 bytes);
2. function `uN(un)` with IPv6-in-IPv6 encapsulation without SRH. This experiment is similar to experiment 1, but the packet size is "artificially" extended, by adding 40 bytes of payload padding, up to the same size of an IPv6 packet with an SRH containing two SIDs (i.e. 158 bytes);
3. function `uN(un)` with IPv6 packets plus a SRH containing two SIDs (158 bytes);
4. function `uN(End)` on IPv6 packets plus a SRH containing two SIDs (158 bytes);
5. function `End` (basic SRv6) on IPv6 packets with an SRH containing two SIDs. Such behavior is considered to be my performance baseline. The other experiments are compared to this one to understand the overhead introduced by the Micro SID header compression mechanism. The packet size for this experiment is 158 bytes.

The detailed results of the above described experiments for the Linux kernel uN implementation are reported in table 3.4. For each experiment, I performed 60 runs with a duration of 10 seconds each. Therefore, each experiment is the average of the results of the 60 runs. The throughput (848.60 kpps) measured in experiment 5 is taken as reference to evaluate the increase or decrease in performance experienced in the other experiments. Indeed, the SRv6 End behavior does not perform any uN operation so that it allows me to find out the impact of the uSID processing with respect to the base SRv6 processing. For each experiment reported in table 3.4, I run the performance tests

#	Function	Encap	PDR@0.5%	Perf. Gain
1	uSID_un	IPv6 in IPv6	869.61 kpps	+2.48%
2	uSID_un	IPv6 in IPv6 (pad)	869.66 kpps	+2.48%
3	uSID_un	IPv6 + SRv6	861.52 kpps	+1.52%
4	uSID_end	IPv6 + SRv6	843.17 kpps	-0.64%
5	End	IPv6 + SRv6	848.60 kpps	———

Table 3.4: *Linux kernel performance assessment for Micro SID.*

to estimate the maximum throughput considering the Partial Drop Rate fixed at 0.5% (PDR@0.5%).

As expected, the processing overhead introduced by the uN behavior depends on which operation is performed and on the packet encapsulation. The IPv6-in-IPv6 encapsulation achieves the highest performance in terms of throughput. The fixed IPv6 header size along with a more efficient parsing are the key factors which increase the overall throughput of 2.48% with respect to the baseline (SRv6 End behavior).

Considering the SRv6 encapsulation, the `uN(un)` performance is slightly better than the performance of the SRv6 End behavior with a measured gain of 1.52% On the other hand, when the `uN(End)` operation is applied on SRv6 packets the measured performance drop with respect to the baseline is 0.64% and thus it could be considered practically negligible.

These results show that the large saving in packet overhead the uSID solution provides, does not reduce performance with respect to standard SRv6 processing.

3.6 Related works

Evaluating software forwarding performance on commodity CPUs requires accurate measurements and analysis. For this purpose, several frameworks have been developed. However, unless [133], none of the work found in the literature has fully addressed the performance of the SRv6 data plane implementation in either the Linux kernel or other software router implementations (e.g., VPP).

In [14], the authors presented an analytical queuing model to evaluate the performance of a DPDK-based vSwitch. The authors studied several characteristics of the DPDK framework, i.e.: average queue size, average loss rate under different arrival loads, etc. In [126], the performance of several virtual switch implementations including Open vSwitch (OVS) [123], SR-IOV and VPP are investigated. This work focuses on the NFV use-cases where multiple VNFs run in x86 servers showing the system throughput in a multi-VNF environment. Next, this work has been extended by the same authors and, this time, they replaced OVS with OVS-DPDK [124] significantly increasing performance for VNFs. However, the all these studies were conducted considering only IPv4 traffic.

In [11], the authors thoroughly explain the main architectural principles and components of VPP including, i.e: kernel bypassing based on DPDK, vector processing, packet batch processing and so on. To validate the high-speed fowarding capabilities of VPP, the authors report some performance metrics such as packet forwarding rate at various vector sizes and different programming techniques used to improve CPU cache utilization. However, this work only analyzes VPP forwarding performance for IPv4

forwarding and does not consider other forwarding types such as IPv6 and SRv6.

Concerning the performance assessment methodology, [114] proposes an algorithm to search for the NDR and the PDR at the same time. The algorithm is called *Multiple Loss Ratio Search for Packet Throughput* (MLRsearch). Actually, it can search for PDRs with different target packet loss ratios at the same time (hence the adjective *multiple*). The fundamental difference between the MLRsearch algorithm and the proposed algorithm discussed in Subsection 3.3.3 is that MLRsearch assumes that the system is deterministic, i.e. the packet loss ratio for a given offered load can be measured without uncertainty in a relatively short amount of time. For this reason, the MLRsearch algorithm does not need to repeat measurements to estimate their standard deviation.

The work in [175] presents a performance evaluation methodology for Linux kernel packet forwarding. The methodology divides the kernel forwarding operations into execution area (EA) which can be pinned to different CPU cores (or the same core in case of single CPU measures) and measured independently. The EA are: i) sending; ii) forwarding; iii) receiving. The measured results consider only the OVS kernel switching in case of single UDP flow.

3.7 Conclusions

In this Chapter, I described the design and implementation of SRPerf, a performance evaluation framework for SRv6 implementations. SRPerf has been designed to be extensible: it can support different forwarding engines including software and hardware forwarding, and can also be extended to support different traffic generators. Moreover, I presented the evaluation methodology for the forwarding performance based on the estimation of the NDR and PDR metrics.

SRPerf has been used for evaluating the performance of the most commonly used SRv6 behaviors in the Linux kernel and VPP as reported in [133]. In this Chapter, I leveraged the SRPerf framework for validating and testing the performance related to the new SRv6 Endpoint behaviors and SRv6 capabilities (i.e. counters) that I introduced in the Linux kernel mainline. I also used SRPerf to test successfully the SRv6 uN behavior coming with the new Micro SID extension to the Network Programming Model. Thanks to this framework, it is possible to continuously monitor and quantify the impact of new developments introduced in a software-based router such as the Linux kernel.

CHAPTER 4

An Efficient Linux kernel Implementation of Service Function Chaining for legacy VNFs based on IPv6 Segment Routing

The work I present in this Chapter is mostly taken from my paper *An Efficient Linux Kernel Implementation of Service Function Chaining for legacy VNFs based on IPv6 Segment Routing*, presented at *5th IEEE International Conference on Network Softwarization (NetSoft 2019)* [106].

4.1 Introduction

Network Operators are facing difficult challenges to keep up with the increasing demand for capacity, the need to support fast service creation and deployment and at the same time the goal of reducing the costs. Network Function Virtualization (NFV) [121] [134] and Software Defined Networking (SDN) represent an answer to these challenges and are changing the way IP networks are designed and operated. Leveraging Cloud Computing principles, NFV moves the traditional data-plane network functions from expensive, closed and proprietary hardware to the so-called Virtual Network Functions (VNFs) running over a distributed, cloud-like infrastructure referred to as NFVI (NFV Infrastructure). The SDN architecture splits the data and control planes and moves the intelligence to the SDN controller. SDN aims at simplifying the introduction of new services and fostering flexibility thanks to the centralized network state view.

The concept of services chaining (also known as Service Function Chaining - SFC [65]) is directly associated with NFV. Actually, the idea of creating a processing path across services pre-dates the NFV concept as stated in [132] and [23]. In fact, service

chaining has been traditionally realized in a static way by putting hardware functions as middle-points of the processing paths and in some cases by diverting the forwarding paths with manual configuration of VLANs stitching or policy routing. However, these "static" approaches come with several drawbacks which are detailed in [132]. In particular, they are intrinsically difficult to scale and hard to reconfigure. On the other hand, the current view of SFC applied to NFV is that it has to be highly *dynamic* and *scalable*. The IETF SFC Working Group (WG) has investigated the scenarios and issues related to dynamic service chaining [132] and proposed a reference architecture [65]. The main logical elements of this architecture are i) Classifiers; ii) Service Functions Forwarders (SFF), iii) the Service Functions, iv) SFC proxies. The Classifiers match the traffic against a set of policies in order to associate the proper Service Function Chain. The SFFs forward the traffic towards the Service Functions or towards other SFFs and handle the traffic coming back from the Service Functions. The SFC framework proposed in [65] does not pose any restriction on the functions that can be chained: they can be both virtualized (VNFs) or physical functions (PFs). For the sake of simplicity, hereafter in the Chapter I will only refer to the virtualized case[1] and I simply use the term VNF instead of Service Function. In this scenario, the forwarding of traffic along a Service Chain needs to be supported by specific protocols and mechanisms that allow the architectural elements to exchange context information. The VNFs can participate in these mechanisms and, in this case, they are called *SFC aware*. On the other hand, the legacy VNFs that do not interact with the SFC protocols and mechanisms are called *SFC unaware*. The SFC proxy elements are needed for the latter type of VNFs. An SFC proxy hides the SFC mechanisms to the SFC unaware VNFs, that will receive and send plain IP traffic.

The IETF SFC WG is considering the Network Service Header (NSH) [131] as a specific solution for the realization of the SFC architecture. In this Chapter, I foster the use of IPv6 Segment Routing (SRv6) (already introduced in Chapter 1, Section 1.1) to implement Service Function Chaining [52], [85] by adopting an approach fully aligned with the network programming model [27].

A relevant subset of the SRv6 [25] and network programming model specifications have been implemented and integrated in the mainline Linux kernel networking stack [86, 87]. In this Chapter, I rely on this existing work and extend it to focus on the Service Function Chaining of legacy VNFs, which are not able to process the SRv6 headers. The support of legacy VNFs is important for Internet Service Providers (ISP) for different reasons: i) it guarantees a feasible migration strategy saving past investments; ii) it facilitates the interoperability and the multi-vendor scenarios, i.e deployments composed by VNFs coming from different vendors; iii) the development of SRv6 aware VNFs requires a new implementation cycle which can be more expensive in the short period.

As introduced above, a proxy element needs to be inserted in the processing chain as a relay mechanism in order to support SRv6 unaware VNFs (see Figure 4.1). The Linux kernel, even in the latest releases, still lacks the functionality to implement such an SRv6 proxy element. Considering the importance of the support of legacy SR-unaware applications in NFV deployments, I designed and implemented an *SR-proxy* solution integrated in the Linux kernel networking stack. I shortly refer to this work

[1] I consider only virtual functions as they can model any sort of packet processing functions without any hardware dependency

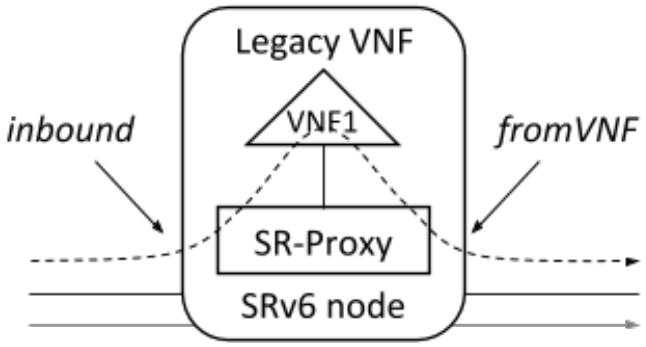


Figure 4.1: *SRv6 NFV Node with SR-Proxy for SR-unaware VNF.*

as SRNK (SR-Proxy Native Kernel). The performance of the proposed solution has been evaluated, identifying a poor scalability with respect to the number of VNFs to be supported within an NFV node in my first design. The issue is actually related to the implementation of Policy Routing framework in Linux. Therefore I provided a second design, enhancing the Linux Policy Routing framework. The performance of the new design is independent from the number of supported VNFs in a node. To the best of my knowledge the only work providing a similar functionality is SREXT [3], but it was designed as an external module not integrated with the most recent SRv6 developments in the Linux kernel.

The content of the Chapter is structured as follows. Section 4.2 introduces SFC based on SRv6 considering both SRv6 aware and unaware VNFs. My proposed design and implementation of SRv6 Proxy to support legacy VNFs in the Linux kernel is described in Section 4.3. In the 4.4 section, I describe in detail how the SRv6 proxy can be set up in Linux. A brief discussion about the testing environment and methodologies for performance analysis are reported in Section 4.5. Section 4.6 covers the performance evaluation of the implemented solutions. Finally, in Section4.7 I discuss about alternative implementations of SRv6 behaviors outside the official SRv6 Linux kernel subsystem, while in Section 4.8 I draw some conclusions and discuss future work.

4.2 SFC Based on IPv6 Segment Routing

In my research activity, I consider the use of SRv6 for SFC, leveraging its scalability properties. Thanks to the source routing approach, SRv6 is able to simplify network operations. Generally speaking, the advantage of approaches based on source routing lies in the possibility to add state information in the packet headers, thus avoiding or minimizing the information that needs to be configured and maintained by the internal nodes. The possibility to interact only with the edge nodes to set up complex services is extremely appealing from the point of view of simplicity and efficiency. This greatly improves the scalability of services based on SR and allows simpler and faster service setup and re-configuration. In [55], the scaling capability of Segment Routing has been demonstrated considering a use case of 600,000 nodes and 300 millions of endpoints.

By exploiting the SRv6 approach, VNFs can be mapped in IPv6 addresses in the *segments* list (also referred to as Segment List or SIDs list) and, in this way, it is possible to represent the VNF chain using this list carried in the Segment Routing Header (SRH).

The SR information can be pushed into the packets using two different approaches,

denoted as *insert* and *encap* modes, respectively. When a node uses the *insert* mode, the SRH is pushed as the next header in the original IPv6 packet, immediately after the IPv6 header and before the transport header. The original IPv6 header is changed, in particular the *next header* is modified according to the value of SRH, the IPv6 destination address is replaced with the IPv6 address of the first SID in the segment list, while the original IPv6 destination address is carried in the SRH header as the last segment of the list. Although the *insert* was present in early draft versions of the network programming model document, it was later removed and is no longer present in RFC 8986 [27]. This choice was motivated primarily by the fact that *insert* mode heavily alters the original IPv6 header. Therefore, I will mainly consider the *encap* mode where the original IPv6 packet is transported as the inner packet of an IPv6-in-IPv6 encapsulated packet and travels unmodified in the network. The outer IPv6 packet carries the SRH header with the segments list. As any tunneling (encapsulation) method, SRv6 introduces overhead the packets. The *insert* mode introduces an overhead of $8 + N * 16[bytes]$, while in the *encap* mode the overhead is $40 + 8 + N * 16[bytes]$ where, in both cases, N is the number of segments carried in the SID List.

An *SR-aware* VNF can process the SRH of the incoming packets and can use it to influence the processing/forwarding of the packets. Such VNFs interact with the node Operating System or with SR modules in order to read and/or set the information contained in the SRH. On the other hand, the *SR-unaware* (legacy) VNFs are not capable of processing the SRv6 SFC encapsulation. In this scenario an *SR proxy* is necessary to remove the SRv6 header and deliver a "clean" IP packet to the VNF. Figure 4.1 provides the reference architecture for a SRv6 NFV node that includes an *SR-unaware* VNF (VNF1 in the Figure). I refer to packets incoming to the SRv6 NFV node that should be forwarded to the VNF by the SR-proxy as *inbound* packets. The SR-Proxy needs to intercept the packets coming out from the VNF and re-apply the SRv6 SFC encapsulation. I refer to these packets as *fromVNF* packets.

In [52], a set of SR-proxy *behaviors* have been defined and among them I mention: i) *static* proxy (also called End.AS behavior); ii) *dynamic* proxy (End.AD behavior); iii) *masquerading* proxy (End.AM behavior). The first two cases (*static* and *dynamic* proxies) support IPv6 SR packets in *encap* mode. The encapsulated packets can be IPv6, IPv4 or L2 packets. The SR proxy intercepts SR packets before being handed to the *SR-unaware* VNF, hence it can remove the SR encapsulation from packets. For packets coming back from *SR-unaware* VNF, the SR proxy can restore the SRv6 encapsulation and update the SRH properly. The difference between the *static* and the *dynamic* proxies is that the SR information that needs to be pushed back in the packets is statically configured in the first case and it is *learned* from the incoming packets in the *dynamic* case. Instead, the *masquerading* proxy supports SR packets traveling in *insert* mode. Since the *insert* mode has been deprecated, then *masquerading* is no longer a very interesting case and therefore I do not discuss it further.

Hereafter, I discuss the operational model and the state information that need to be configured and maintained in the SRv6 NFV nodes. Figure 4.2 illustrates a SRv6 based NFV domain, in which the VNFs are hosted in different NFV nodes. The packets to be associated with VNF chains are classified in *ingress* nodes, where the SR encapsulation is added. A network operator willing to use SRv6 SFC chaining for *SR-unaware* VNFs, will first need to associate VNFs to Segment IDs (SIDs) in the hosting SRv6

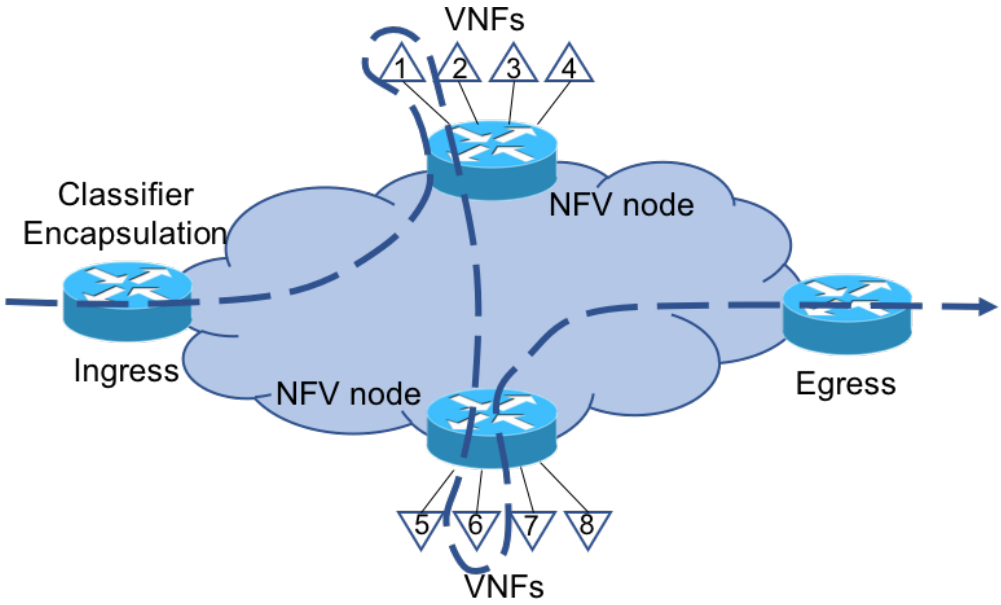


Figure 4.2: *SFC scenario.*

NFV nodes. I recall that a SID is represented by an IPv6 address. Each SRv6 NFV node has a pool of IPv6 addresses (prefixes) that are available to be used as SIDs for its VNFs. These prefixes are distributed using regular routing protocols, so that the reachability of all VNFs is assured. The association of the IPv6 address SID to a VNF is a configuration operation to be performed in the SRv6 NFV node and it binds the SID to the virtual interface that connects the SR-proxy to the VNF. This operation is performed when a legacy VNF is created in a NFV node. The corresponding state information is used in the *inbound* direction, when packets directed to the VNF are processed by the SR-proxy. The second step is to configure a VNF chain across the VNFs that are running over the SRv6 NFV nodes. The VNF chain will be applied to a packet by inserting a SID list in the IPv6 SR header in the ingress node. Therefore, the classification of packets and the association with the SID list has to be configured in the ingress node. Each NFV node which runs a legacy VNF needs the proper information to process the packets in the *fromVNF* direction. This is done differently for the respective types of proxy. In the *static* proxy case (End.AS behavior), the state information needed to process the packets coming from the VNF is done by statically configuring the SR-proxy with the SID list to be re-inserted in the packet. Both the *dynamic* proxy (End.AD behavior) and the *masquerading* one (End.AM behavior) have the good property that they do not need to be configured when a new chain is added. The dynamic proxy "learns" the SID list from the packets in the *inbound* direction (and so it saves state information). The *masquerading* proxy does not even need to save the state information as the SID list is carried along with the packet through the legacy VNF (which has to be IPv6 and needs to accept the SRH header without interfering with it). Table 4.1 compares the different SR proxy behaviors.

In this Chapter, I focus on the design and in-kernel implementation of the SR *dynamic* proxy as it represents the most versatile solution (being able to support legacy VNFs working with IPv6, IPv4 and L2 packet) and it offers a simple operational model.

Table 4.1: *Comparison of SR proxy behaviors*

	End.AD	End.AS	End.AM
Generate traffic	Yes	Yes	No
Modify packets	Yes	Yes	No
Stateless	No	No	Yes
State-config	Auto	Manual	N/A
Traffic supported	IPv4/IPv6/L2	IPv4/IPv6/L2	IPv6

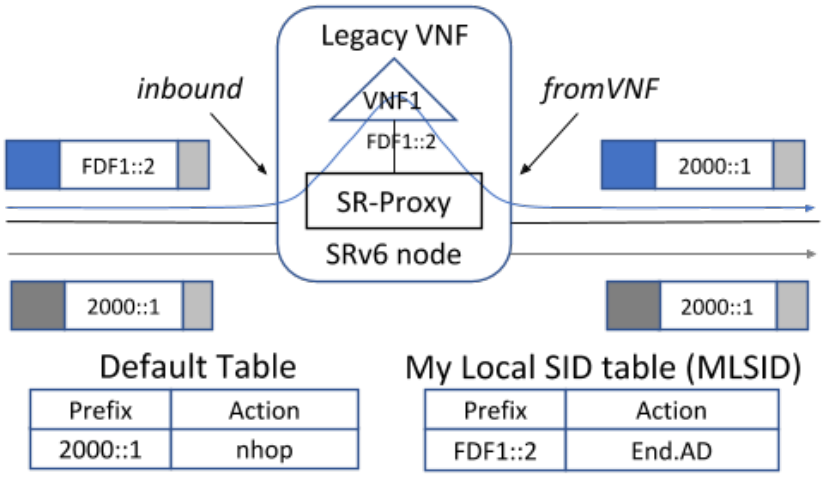


Figure 4.3: *SRv6 node processing.*

4.3 Design of the SRv6 Proxy

Hereafter, I describe the design and implementation aspects of the SR-proxy. I start with Subsection 4.3.1 in which I provide a brief introduction of general concepts that will be extensively used in this Chapter as well as the State-of-the-art for in-kernel SR-Proxy solutions. In Subsection 4.3.2, I present the first implementation of my SR-Proxy, referred to as SRNKv1 (Native Kernel v1) and I elaborate on its operations. In Subsection 4.3.3, I analyze the performance issues of SRNKv1 and, then, I come up with a second design and implementation (SRNKv2) discussing its performance improvements.

4.3.1 General Concepts and State-of-the-art

SR-proxy in the network programming model

In Chapter 2, I discussed in detail how the SRv6 network programming model has been implemented in the Linux kernel and how the processing model can be exploited to perform generic operations on packets. The purpose of my work is to extend the implementation of the SRv6 network programming model (`seg6local` LWT infrastructure, Section 2.1.3) currently implemented in the Linux kernel to support the *dynamic* proxy (End.AD behavior). Figure 4.3 shows the processing of a SRv6 node where a legacy VNF is deployed. With reference to Figure 4.3, I explain the details of SRv6 processing in an NFV node hosting an SR-proxy. For the packets in the *inbound* direction, the SR-proxy classifies the packets based on the IPv6 destination address, decapsulates them as needed and forwards to the proper interface towards the VNF. For the packets

in the *fromVNF* direction (i.e. sent back by the *SR-unaware* applications), the SR-proxy needs to restore the SRH header after the identification of the interface from where the packets are coming. Looking at Figure 4.3 an *inbound* packet having as destination address which does not correspond to a VNF (e.g. `2000::1`) is simply forwarded by the node over an outgoing interface, looking at the default routing table. Conversely, a packet having `FDF1::2` as IPv6 destination address (and active *segment* in the segment list) is matched by the node in *My Local SID Table*, hence the *SR-proxy* behavior is applied and the packet is forwarded to the VNF1. When considering a packet coming from the legacy VNF1, the proxy correctly restores the SRv6 header and delivers it to the IPv6 processing of the node that will forward to the next hop. Note that *My Local SID Table* and the normal routing table does not need to be separated, this is actually an implementation aspect. In the current Linux kernel implementation the SID entries can be inserted in any routing table, therefore also in the default routing table.

State-of-the-art - SREXT module

The SREXT module ([3]) is an implementation of the SRv6 network programming model. As an SR-proxy, SREXT handles the processing of SR information on behalf of the SR-unaware VNFs, which are attached using two interfaces. SREXT provides an additional local SID table which coexists with the one maintained by the Linux kernel. The SREXT module registers itself as a callback function in the *pre-routing* hook of the Linux Netfilter [119] framework. Since its position is at the beginning of the Netfilter processing, it is invoked for each received IPv6 packet. If the destination IPv6 address matches an entry in the local SID table, the associated behavior is applied otherwise the packet will follow the normal processing of the routing subsystem.

A secondary table (the so-called "srdev" table) is used by SREXT for correctly executing the processing of the *inbound* and *fromVNF* packets. As regards the former, once the packet has passed the sanity check and the SRv6 behavior has been applied, SREXT stores in this table the *fromVNF* interface (where SREXT will receive back the packet from the VNF), the applied behavior, the original IPv6 header and its SRH. On the *fromVNF* side, the receiving interface is used as a look-up key in the table "srdev", if an entry is found the headers are re-added (IPv6 control traffic like NDP is dropped) and finally the packet will go through the kernel IP routing sub-system for further processing. A new *cli* has been implemented for controlling SREXT behaviors and showing its tables and counters.

4.3.2 SRNKv1

In this section, I present the design of my first kernel implementation of the *dynamic* proxy (End.AD behavior), referred to as *SRNKv1*. Most of the following design choices apply also to the static proxy (End.AS behavior), which can be seen as a by-product of the the *dynamic* proxy implementation. In order to simplify the discussion I just mention the *dynamic* proxy in the following paragraphs and in the images. *SRNKv1* design relies on two distinct tunnel (LWT) instances which manage respectively the *inbound* and *fromVNF* traffic. For each tunnel instance, state information is maintained in order to correctly perform the proxy operations. In particular, the *inbound* processing needs an entry on the *My Local SID Table* and uses a shared hashtable to store the headers that have to be restored during the *fromVNF* processing.

As regards the traffic coming from the legacy VNF, a policy routing entry for each VNF is necessary to classify the packets, a routing table with a default route pointing to the LWT is used for the VNF and finally the shared hashtable is used to read the headers stored previously by the *inbound* processing. Figures 4.4 show a high-level view of the processing inside a SRv6 enabled node and how IPv6 routing network subsystem interacts with the SRv6 *dynamic* proxy implementation.

Inbound processing

As soon as an IPv6 packet arrives at `eth0` of the NFV node it enters within the Linux kernel networking stack. After passing the `pre-routing` stage, the kernel tries to look up the route with the longest prefix matching the active *segment* of the packet. Due to policy-routing settings, the kernel looks first at *My Local SID Table* and if no matching route has been found, it considers the other tables and possibly moves on the next stages of the processing (`input` or `forward`). Figure 4.4a shows this process, the packet destination address matches with prefix `sid1` and the corresponding route is used. Therefore, the Linux kernel executes the processing function associated with the route: the *inbound* End.AD operation. The inbound End.AD operates in three different stages: i) it pops the outer IPv6 and SRv6 headers from the incoming packet; ii) it updates the SID pointer of the SRv6 header to select the next one and stores such retrieved headers into a *per-net namespace* hashtable data structure; iii) finally it sends out the decapsulated IPv6 plain packet to its designated legacy VNF.

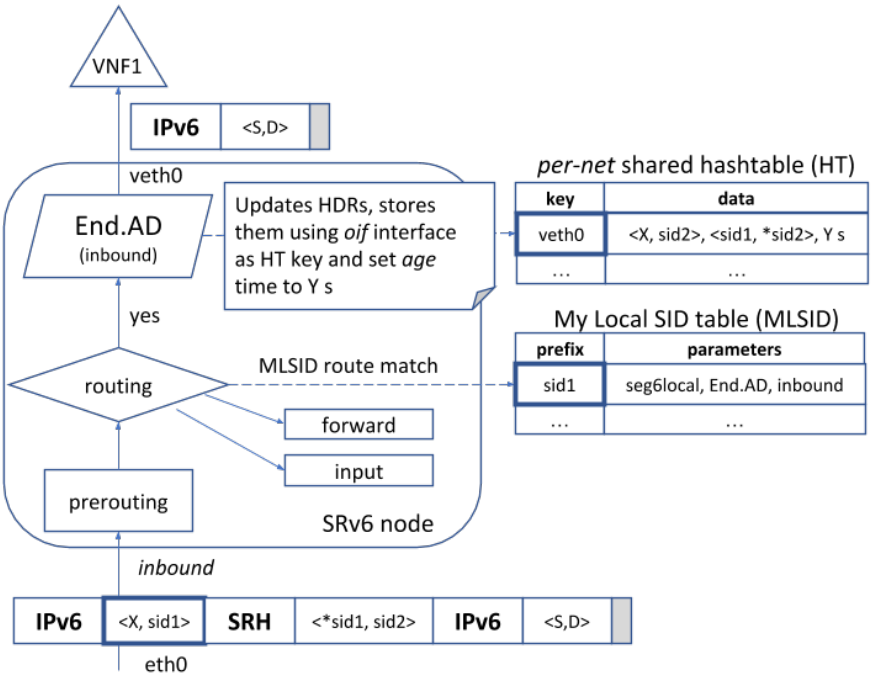
Removed headers at step ii) are indexed in the hashtable by using the identifier of packet outgoing interface, the one used to communicate with the legacy VNF (`veth0` in Figure 4.4a). Due to the necessity of sharing IPv6 and SRv6 headers between *inbound* and *fromVNF* processing, the choice of storing them within an external shared data structure resulted to be the right solution. This design simplifies the access pattern to the stored data, as well as it increases performance. Indeed, the hashtable is well suitable to support fast data retrieval with a very low computational cost and, ideally, it is independent with regard to the number of stored entries.

From a configuration point of view, the *inbound* processing just relies on the plain IPv6 routing through *My Local SID Table*: the new route is added with the `ip -6 route add` command of the `iproute2` suite, by also specifying the behavior to be activated in the parameters of the command. The details on the configuration commands are described in Subsection 4.4.1.

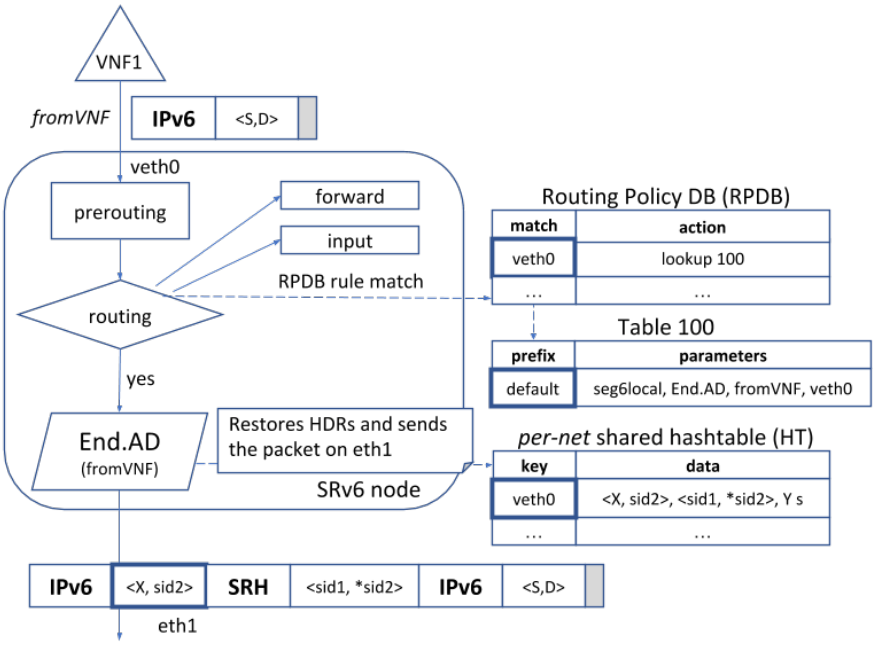
Auto-learning process

The auto-learning process consists in learning the information related to the VNFs chain (i.e., the list of segments in the SRv6 header) from the inbound packets, without the need of a static configuration. The learned information is saved in a shared hashtable. I have introduced a specific attribute[2], named `age`, to control the rate at which the shared hashtable can be updated. This attribute can be set during the setup of the LWT routing entry in *My Local SID Table*. When different from 0, the `age` attribute represents the minimum interval (in seconds) between two write operations in the shared hashtable for the same VNF. Setting the `age` to 1 second corresponds to a maximum reconfiguration delay of 1 second for a NFV node when the VNF chain is changed by an ingress node

[2]I refer to the *attribute* of an SRv6 behavior also as *parameter*.



(a) *SR-proxy Inbound processing (SRNKv1).*



(b) *SR-proxy FromVNF processing (SRNKv1).*

Figure 4.4: *SRNKv1 design*

and this is the default I used in my experiments. If `age` equals 0, the hashtable is updated for every inbound packet, providing the fastest possible reconfiguration time for a VNF chain. In the performance evaluation section, I have analyzed the performance cost for the continuous updating of the shared hashtable with respect to the default minimum reconfiguration delay of 1 second. The `age` attribute registers the last time the headers have been updated and it is used also to determine, when a packet is received,

if it is the time to replace stale data with new fresh one. The auto-learning operation is performed only during the *inbound* processing. The learned information are retrieved during the *fromVNF* processing using the incoming interface[3] of the packet to rebuild the whole SRv6 packet ready for being forwarded over the network.

Setting properly the `age` attribute has an important impact on the performance of the system and a proper trade-off is necessary according to the use case to be supported. In a shared-memory producer-consumer context, it is possible to identify the *inbound* processing as the content producer, and the *fromVNF* one as the consumer. Indeed, the former is in charge of keeping the global hashtable up-to-date, while the latter accesses the structure for retrieving the headers. Considering this model, the aging attribute can be seen as the upper-bound of data production/refresh rate. By setting it to the maximum limit, is it possible to prevent overloading of the SRv6 NFV node caused by high-rate writing in the shared memory.

This problem is particularly noticeable in all of those systems based on multi-core architectures: the Linux kernel networking stack allows to assign the received packets to all available computing units in order to process them in parallel and to support high data rates. However, this means that several End.AD processing operations may occur at once and, potentially, they may involve updating the same IPv6 and SRv6 headers. Very frequent and simultaneous shared memory updates by multiple CPUs can lead to conflicts that can negatively affect the overall performance of the system. For all these reasons, small values for the `age` attribute make the system more responsive to chain (SRv6's segment list) changes, but on the other side they can push heavy and unnecessarily load to the SRv6 NFV node due to high data refresh rate.

FromVNF Processing

The *fromVNF* LWT is meant to work in tandem with its *inbound* counterpart. The association between the *fromVNF* packets and the tunnel cannot rely on the IPv6 Destination Address carrying a SID as it happens for the *inbound* packets. The match needs to be uniquely performed, based on the incoming (internal) interface between the VNF and the NFV node. For each VNF to be supported, I add an entry in the IPv6 Routing Policy DB (also known as IPv6 rule, as the command used to configure it is `ip -6 rule`). The rule points to a different routing table for each VNF, in which there is only a default route, pointing to the lightweight tunnel associated with the VNF. This means that for N VNFs, there will be N rules and N routing tables. Figure 4.4b provides a representation of the described *fromVNF* processing.

In the following, I analyze the motivations behind this design choice. The *fromVNF* LWT can not be tied to any route with a specific prefix because the IPv6 packets sent by VNF can use any destination address and do not have any relationship with the SIDs. Moreover, each End.AD *fromVNF* tunnel expects to receive traffic by its own layer-2 interface (veth0 in Figure 4.4), with no regards about the IPv6 destination address of the packets. This means that, in order to apply the *fromVNF* processing function to an incoming packet, the SRv6 NFV node has to retrieve the route that points to the right LWT using only the identifier of the interface where such packet was received. As a consequence of this, the *fromVNF* End.AD design has to deal with: i) the problem of

[3]The current implementation of the *dynamic proxy* assumes that the same interface is used to interact with VNF in the two directions.

designating an IPv6 prefix to be used for creating a route pointing to a custom processing function (LWT), and ii) the issue of steering incoming traffic received on a specific interface through such as route.

The first issue can be easily solved by using as route prefix the *any address* which is often indicated by "::". Generally, the *default route* is selected by the routing algorithm when the IPv6 destination address can not be managed by any other route. However, this approach gives rise to a new problem. Indeed, creating a LWT on a *default route* has the side effect that no more than one VNF can be handled by the SRv6 node using a single table. Moreover, control traffic that transits through the SRv6 node and for which there are no given explicit routes may be wrongly handled by the LWT installed on the *default route*. Thankfully, this problem can be easily solved by installing every *default route* into a different IPv6 routing table and creating, for each of these, an entry in the IPv6 Routing Policy DB (also known as IPv6 rule, as `ip -6 rule` command is used to configure it). Such as rule is meant to instruct the IPv6 networking subsystem to perform route look-up on a specific table based on a specified match. The usage of an IPv6 policy route solves also the issue ii) as, at this point, it is possible to use the *fromVNF* interface (`veth0` in the above example) as match and a `goto-table N` as action predicate. In this way, an interface can be related to a specific *default route* that is attached to a LWT.

Figure 4.4b shows a high-level overview of the proposed solution with the *fromVNF* LWT tunnel integrated in the IPv6 routing network subsystem. Whenever a plain IPv6 packet, sent by VNF, arrives at SRv6 NFV node, it is handled by the Linux kernel networking stack. After passing the `pre-routing` stage, the kernel tries to determine the right processing of the packet. It invokes the route look-up operation, but this time the routing algorithm finds first an entry in the RPDB of the node and does not consider IPv6 destination address at first. Indeed, thanks to custom IPv6 rules (one for each *fromVNF* tunnel) the routing algorithm is capable of retrieving the IPv6 table tied to the incoming interface of the packet. At this point, the routing algorithm makes use of this table to find out the route that matches with the received packet. In this specific case, the routing algorithm selects and returns the only route available, the `default` one, that is attached to a specific End.AD tunnel. Once the plain IPv6 packet has been received by the *fromVNF* processing function, it leverages the identifier of the incoming interface of the packet to search for the popped IPv6 and SRv6 headers within the hashtable. If a result is found, the processing function forges a new packet and sets the headers of that packet with the ones that were just retrieved. The plain IPv6 packet is encapsulated into the newly created one and then the whole packet is delivered towards its destination. This concludes the job of the *fromVNF* LWT tunnel processing operation.

Details on configuration for *fromVNF* processing are given in Subsection 4.4.1.

End.AS design

The End.AD differs from the End.AS just in the way the stored headers are managed. The End.AS behavior is a simplification of the End.AD because it does not need to deal with the auto-learning process. Indeed, it uses chain information which has been saved once during the behavior configuration. The list of segments does not change during the entire life of the End.AS instance unless it is first deleted and then saved with new headers values.

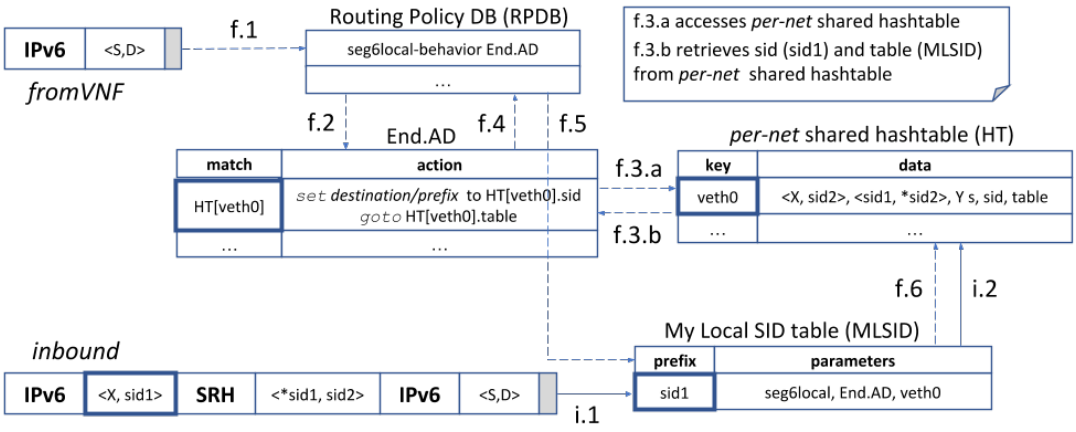


Figure 4.5: *SRNKv2 design.*

4.3.3 SRNKv2

After the implementation of SRNKv1, I critically revised its design, by identifying the following main shortcomings: i) two LWTs are used for the two directions *inbound* and *fromVNF* related to the same VNF; ii) a different routing table needs to be instantiated for each VNF so that the correct LWT can be associated with the *fromVNF* packets; iii) the use of the Linux Policy Routing framework implies to sequentially scan a list of *rules* to identify the VNF interface from which a packet is coming and associate a specific routing table. In particular, the first two shortcomings correspond to a waste of memory resources, while the third one to a waste of processing resources (see Section 4.6).

The revised design of SRNKv2 is shown in Figure 4.5. The most important improvement is an extension to the Linux Policy Routing framework, in order to avoid the linear scan of the list of rules to match the VNF interface (i.e. one rule for each VNF). A new type of IPv6 rule, called SRv6 extended rule, is added and used in the *fromVNF* processing (f.1). The new rule (indicated as seg6local-behavior End.AD in the Figure 4.5) is added to the Routing Policy DB. The *selector* of this rule performs the lookup of the packet incoming interface (f.2) into the shared hashtable that includes all the VNF interfaces handled by the *dynamic* proxy. In this way, it is possible to understand if the packet is coming from a VNF and to retrieve the needed information (f.3.a, f.3.b). The SID associated to the VNF is used to perform the search (f.5) into the *My Local SID Table*, which will return the associated LWT.

In the new design, there is actually a single LWT for the two directions. In fact, the function that is executed when a routing entry points to the tunnel is able to understand whether the packet belongs to the *inbound* or to the *fromVNF* direction and behave accordingly. Thanks to the lookup in the shared hashtable, which allows retrieving the SID associated with the VNF, it is no longer required to have a separate routing table for each VNF. Finally, SRNKv2 also simplifies the configuration.

In Subsection 4.4.2, I provide further details on the new configuration procedures for the SRNKv2.

4.3.4 Implementation of other SR proxy types

In addition to the implementation of the SR *dynamic proxy*, I have already implemented the *static proxy* behavior. Actually, this is simple extension of the *dynamic* one where: i) I developed the command line *static* configuration on `iproute2` tool-suite; ii) I disabled the "learning" capability. In principle, the SR proxy solution can be adapted also to implement the *masquerading proxy*, however, I left task for a future work.

4.4 SR Proxy configurations

4.4.1 SRNKv1 configuration

Hereafter, I report the configuration procedure of the SRNKv1 implementation.

Inbound processing

The *inbound* End.AD tunnel does not require any particular configuration for the Policy Routing subsystem. To instantiate an *inbound* End.AD tunnel that can handle packets destined for the VNF with SID *fdf1::2*, the following command is used:

```
$ ip -6 route add fdf1::2/128 encap seg6local           \
    action End.AD chain inbound oif veth0 nh6 fdf1::2   \
    age 5 dev veth0
```

It is easy to identify in its structure three different parts:

- `ip -6 route add fdf1::2/128` is used to add the route `fdf1::2` in the IPv6 routing tables;
- `encap seg6local` is used for specifying to the IPv6 network subsystem to create a `seg6local` tunnel which handles packets for `fdf1::2`
- `action End.AD chain inbound oif veth0 nh6 fdf1::2 age 5` is used for specifying the processing function to be performed by the `seg6local LWT` along with several configuration attributes.

In this sub-command, the action is defined as *End.AD* and the direction of the data flow is towards the VNF (`chain inbound`). As I explained in Subsection 4.3.2, each packet that comes into this tunnel is subjected to an outer IPv6 and SRv6 headers decapsulation. The `nh6` param is used to inform the inbound tunnel about the next hop to which each packet has to be sent, and in this case is the SID of the legacy VNF. Finally, the `age` is used to enforce the refresh time (set to 5 seconds) for the auto-learning feature.

FromVNF processing

The *fromVNF* End.AD tunnel has to perform the inverse operation realized by the *inbound* counterpart. It has to restore the IPv6 and SRv6 headers using only the incoming interface of the packet as key search. At this point, I need to instruct the network subsystem on how it should send traffic to the right *fromVNF* tunnel. Using the `ip -6 rule` tool, I am able to manipulate the Routing Policy DB of the nodes and issue

a command that informs the system to make use of a given IPv6 routing table when traffic arrives at some specific interface. The *ip rule* command is the following:

```
$ ip -6 rule add iif veth0 table 100
```

After the execution of this command, every time a packet arrives at the ingress interface `veth0`, the routing system will try to find a route in `table 100` that matches with the destination address of that packet.

Instead, to create a *fromVNF* End.AD tunnel on SFF/SR node with the purpose of managing packets coming from the legacy VNF at interface `veth0`, the following command is used:

```
$ ip -6 route add default encap seg6local \
    action End.AD chain fromVNF iif veth0 \
    dev veth0 table 100
```

Also in the case of *fromVNF* tunnel creation, the `ip` command can be seen decomposed into three different parts:

- `ip -6 route add default` is used to add the `default route` "::" in the IPv6 routing table `100`;
- `encap seg6local` is used for specifying to the IPv6 network subsystem to create a `seg6local` tunnel which handles packets destined for `default` address;
- `action End.AD chain fromVNF iif veth0` is used to specify the attributes of the behavior that is intended to be created.

The action is `End.AD` and the direction of the data flow is specified by the chain attribute which is set to `chain fromVNF`. The `iif` keyword indicates packets coming the from interface `veth0`. This means also that the tunnel is allowed to listen for incoming traffic only from the interface specified by `iif`. If it receives packets from another interface, those are discarded automatically.

4.4.2 SRNKv2 configuration

Hereafter, I report the configuration procedure of the SRNKv2 implementation.

Tunnel creation

In my second design (Subsection 4.3.3) I have introduced the notion of bi-directional tunnel within the definition of the End.AD behavior. The creation of the desired behavior can be achieved through the following command:

```
$ip -6 route add fdf1::2/128 encap seg6local    \
    action End.AD oif veth0 nh6 fdf1::2 age 5   \
    dev veth0
```

As it is possible to appreciate, the command closely resembles the one that I have described in the Subsection 4.4.1 to setup the *inbound* tunnel. However this time, I am able to create only one tunnel that manages the *inbound/fromVNF* processing for a given VNF.

With reference to the scenario depicted in Figure 4.5, traffic that arrives at SFF/SR node with destination the VNF's SID (`fdf1::2`) on any interface except `veth0` is sent to `seg6local LWT` associated with the End.AD behavior for decapsulation purposes (*inbound* processing). After that, packets are delivered to VNF using the output interface `veth0`. On the other side, traffic that arrives from the interface `veth0` has to be redirected to the right End.AD tunnel for the encapsulation (*fromVNF* processing). To accomplish that, the IPv6 rule, described in the next paragraph, is used.

IP rule configuration

The idea behind RPDB configuration relies on the ability to forward packets from a specific incoming interface to the associated End.AD tunnel and process them similarly to the *inbound* processing. With this idea in mind I have designed, and implemented the changes described in Subsection 4.3.3 and I came up with the following `ip rule` command to redirect traffic from VNFs towards the right End.AD tunnels (*fromVNF* processing):

```
$ ip -6 rule add seg6local-behavior End.AD
```

This command adds a new rule to the RPDB of the node. Specifically, `seg6local-behavior End.AD` is used to indicate the local SRv6 behavior that should be taken into account when the rule is picked up. If the priority of the above rule is not superseded by the priority of other rules, the extended RPDB is able to compare incoming interface of the packet with the `oif` of the existing LWTs through the shared per-netns hashtable. This means that: if the *inbound* interface is equal to (exactly) `oif`, the patched IP rule subsystem gets the correspondent VNF's SID which is then used as the destination address for the packet (in place of the real's one) during the IPv6 routing lookup on "My Local SID Table". As soon as the route is resolved, the associated End.AD tunnel is also retrieved and it is exploited for applying *fromVNF* operations on the incoming packet. Otherwise, if the incoming interface of the packet does not belong to any End.AD instance, the packet is treated as usual and the route lookup is performed, by default, on the destination address.

4.5 Testing Environment

4.5.1 Testbed Description

In order to test both SRNKv1 and SRNKv2 solutions, I set up the same testbed that I have already described and explained in details in section 3.4, Chapter 3. For the sake of clarity, in the following I just briefly recall the testbed architecture. The testbed comprises two nodes denoted as Traffic Generator and Receiver (TGR) and System Under Test (SUT). Such nodes are connected back-to-back as depicted in Figure 3.1. The TGR generates traffic towards the SUT node which is processed according to the forwarding function to be tested and, then, packets are sent back to the TGR node. The

testbed is deployed on Cloudlab and the two machines are bare metal servers whose specs are reported in Table 3.1.

The SRPerf framework, described in Chapter 3, is deployed on both TGR and SUT. Doing so, I am able to schedule and run performance tests in a very predictable and repeatable manner. TRex is used as a traffic generator on the TGR and is driven by SRPerf, while on the SUT is running a compiled version of Linux kernel 4.14 patched with End.AD proxy behavior implementations (namely SRNKv1 and SRNKv2). The SUT comes also with a modified version of `iproute2` tool-suite, which enables the configuration of the *dynamic* proxy.

4.5.2 Methodology

I extended the SRPerf framework to support the SR-proxy. In particular, I modified the CFG scripts with the purpose of adding the needed procedures for configuring the SR-proxy on the Forwarder. Along with that, I also provided a new Experiment CFG file for the Orchestrator in which I described the SR-proxy experiment indicating the name, the algorithm to be used for evaluating performance tests and the packet type. Regarding the latter, I considered an IPv6 UDP packet encapsulated in an outer IPv6 packet. The outer packet has an SRH with a SID list of two segments. The first SID points to the SR-unaware VNF running in the SUT, the second SID corresponds to the Receiver interface of the TGR node from the point of view of the SUT. Regarding the packet size, I have followed the indications from the Fast Data I/O Project (FD.io) Continuous System Integration and Testing (CSIT) project report [35]. In particular, the inner IPv6 packet has an UDP payload of 12 bytes, corresponding to 60 bytes at IPv6 level. The SR encapsulation adds 40 bytes for the outer IPv6 header, and 40 bytes for the SRH with two SIDs. The Ethernet layer introduces 18 bytes for Ethernet header and CRC, plus 20 bytes at the physical layer (preamble, inter frame gap). This packet layout has been dumped in a file that is used by the SRPerf framework to evaluate the maximum throughput that can be processed by the SUT. On that note, I measure throughput, considered as the maximum packet rate, in packet per second (pps) where the Packet Drop Ratio (PDR) threshold, in my experiments, is fixed at 0.5%

4.6 SR-proxy performance analysis

In Figure 4.6, I report the performance characterization of my solution (SRNKv2) compared with a baseline reference (End) and with the pre-existing solution (SREXT). The traffic pattern used for the characterization has been described in subsection 4.5.2. In the baseline scenario, I did not configure any SRv6 behavior on the NFV, which is able to simply forward *inbound* and *fromVNF* packets on the basis of the IPv6 destination addresses. On the other hand, the VNF is SRv6 aware and performs the so-called SRv6 *End* behavior (for this reason the scenario is called End). In the End behavior, a node receives the packets destined for itself and advances the pointer of the segment list to the next segment, updating the IPv6 destination address. As a result, in the baseline scenario the SUT performs two regular IP forwarding operations (each one with a lookup in the routing table) and one SRv6 End behavior (which include a lookup for the incoming SID, an SRv6 header processing and a lookup for the next segment). In the SRNKv2 case, the SUT performs a routing lookup for the incoming SID, it de-

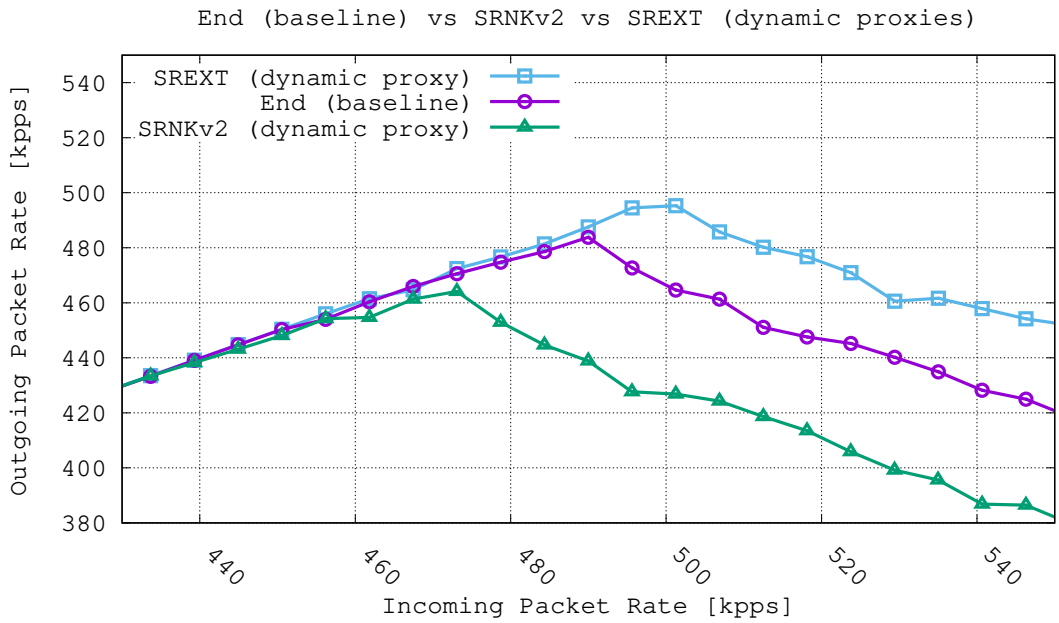


Figure 4.6: *Comparison of End (baseline) vs SRNKv2 vs SREXT.*

capsulates the packet according to the *dynamic* proxy behavior and forwards it to the VNF. The VNF performs a plain forwarding operation (routing lookup) on the inner packet. Moreover, the match on the incoming interface is performed in the NFV node when receiving the packet. The packets are re-encapsulated after retrieving the proper header and finally an IPv6 forwarding operation is performed. The SREXT operations are similar to the ones in the SRNKv2 scenario. The difference is that the matching on the *inbound* packets is not performed in the Linux IPv6 forwarding/routing but the packets are captured by netfilter, in the pre-routing phase. Therefore, the regular forwarding operations are skipped, leading to a higher performance. The PDR@0.5% for SRNKv2, baseline and SREXT are reported in Table 4.2. The SRNKv2 implementation, which also performs decapsulation and re-encapsulation of packets, shows only a 3.7% performance degradation with respect to the baseline forwarding. The SREXT module, which skips the Linux kernel routing operations by capturing packets in the pre-routing hook has a forwarding performance boost of 2.4% with respect to baseline forwarding.

Table 4.2: *Throughput (PDR@0.5%) in kpps.*

SRNKv2	Baseline (End)	SREXT
444.2	461.1	472.3

Figure 4.7 analyzes the poor scalability of my first design (SRNKv1) based on the regular Linux Policy Routing framework. I report the PDR@0.5% versus the number of Policy Routing rules that are processed before the matching one. It is worth noting that the number of rules corresponds to the number of VNFs to be supported and such rules are scanned sequentially until the matching one. Therefore, the performance with N rules can be read as the worst case performance when N VNFs are supported, or as the average case with $2N$ VNFs. A linear degradation of the performance with the

number of rules is clearly visible, for example when there are 80 rules the PDR@0.5% is 28.4% lower than the PDR@0.5% for SRNKv2 or for SRNKv1 with a single rule (for 160 rules the PDR@0.5% is 50.6% lower).

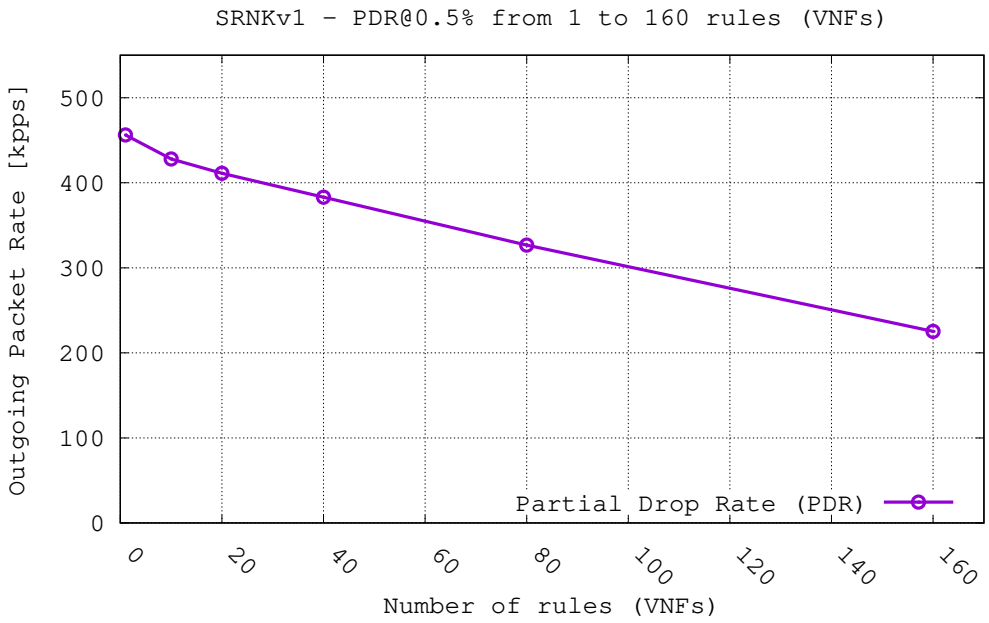


Figure 4.7: *PDR@0.5% vs. number of rules in SRNKv1.*

Regarding the impact of the auto-learning feature of the *dynamic* proxy, in all the experiments shown so far I have evaluated the SRNK performance by setting the `age` attribute to 1 second (hence limiting the update rate to one update/s). However, I run an experiment by setting it to 0 (no limitation on the update rate, so that the VNF chain is updated for each incoming packet). Under this condition, I was not able to consistently achieve a delivery ratio higher than 0.99 even for low packet rates. Therefore, I initially evaluated the PDR@2% (392 kpps) with the *aging* time set to 0 and, then, the PDR@0.5% (444.2 kpps) where the *aging* time was left to the default value (1 second). By comparing those results, I estimated a decrease of performance not less than 11% for updating the VNF chain at the highest possible rate (i.e. for every incoming packet).

Finally, I analyzed the cost of performing the interface lookup with the new extended SRv6 policy rule, separately from the decapsulation and encapsulation operations which are executed in the SRNKv2 scenario. There are two motivations for this analysis. First, the policy rule needs to be executed for all IPv6 packets, so it introduces a performance degradation also for non-matching packets that need to be evaluated. The second reason is that the proposed mechanism could be reused in scenarios with multiple policy rules based on the incoming interfaces. This would require an extension to the Linux Policy Routing framework, the performance evaluation is a part of the cost-benefit analysis for this extension. For this performance analysis, I start from the baseline (End) scenario in which the packets are only forwarded in the NFV according to plain IPv6. I consider two scenarios: i) *Ext. SRv6 Rule* in which an extended SRv6 rule is added with no matching interfaces, the rule will be checked for all *inbound* and *fromVNF* packets; ii) *80 Plain Rules* in which 80 rules to match an

interface are added (with no matching interfaces), these 80 rules will be checked for all *inbound* and *fromVNF* packets. The PDR@0.5% is reported for the baseline and the described scenarios. The performance degradation in the packet forwarding for adding the lookup with the extended SRv6 rule is only 2.7%. On the other hand, adding 80 policy rules implies a big degradation of the forwarding performance, which becomes 28.1% smaller.

Table 4.3: *Extended SRv6 policy rule performance.*

	Baseline	Ext. SRv6 Rule	80 Plain Rules
PDR@0.5% [kpps]	461.1	448.5	328.0
Degradation	-	1.2%	28.1%

4.7 Other SR-proxy implementations

Starting from kernel 4.10, Linux supports SRv6 processing including also the implementation of several SRv6 Endpoint behaviors. However, at the time of writing there is lack of support for proxy behaviors. As already mentioned in Section 4.3.1, the SREXT provides a complementary implementation of SRv6 in Linux based nodes. A specific shortcoming of the SREXT module is that it does not support the Linux network namespaces. Therefore, it cannot coexist with the frameworks and tools that rely on network namespaces (e.g. Linux Containers, Dockers, etc). More generally, a module cannot directly access most helper functions and internal kernel structures. For this reason, it is usually necessary to reimplement them from scratch with the risk of developing inefficient solutions and introducing bugs into the code. The goal of the SRNK implementation is to be integrated in the Linux kernel mainline, so that it can evolve and be maintained together with the Linux kernel.

Another SRv6 implementation is included in the VPP (Vector Packet Processing) [176] platform, which is the open source version of the Cisco VPP technology. VPP supports most of the SRv6 behaviors defined in [27] including also the *dynamic* proxy behavior. As reported in [35], the forwarding performance of VPP is in general very high. For example a NDR (No Drop Rate) of around 5.5 Mpps is reported for a *dynamic* proxy setup similar to SRNK. The VPP performance evaluations are not directly comparable with my measurements, because the former focus only on SR-Proxy operations while the latter also take into account processing within the VNF. In any case, I believe that with comparable configurations, VPP will outperform my implementation (as VPP also outperforms Linux kernel forwarding). However, there is still value in enhancing SRv6 functionality in the Linux kernel as I have proposed, since VPP is not ubiquitously deployed and there may be scenarios where it is easier to use a in-kernel based feature than to depend on an external framework.

4.8 Conclusions

In this Chapter, I have described the design and implementation of a *dynamic* proxy mechanism to support Service Function Chaining based on IPv6 Segment Routing for legacy VNFs, a use case of great importance for service providers. The proposed solution is released as open source and extends the current Linux kernel implementation of

the SRv6 network programming model. The SRv6 proxy has been perfectly integrated in the Linux ecosystem, for example it can be easily configured through the well known `iproute2` user space utility. My plan is to submit the described solution to the Linux kernel mainline.

I have thoroughly analyzed several performance aspects related to my implementation of the *dynamic* SRv6 proxy. I went through two design and implementation cycles, referred to as SRNKv1 and SRNKv2. I identified a scalability issue in the first design SRNKv1, which has a linear degradation of the performance with the number of VNFs to be supported. The root cause of the problem is the Linux Policy Routing framework. The final design SRNKv2 solved the problem, by introducing a new type of rule in the Policy Routing framework. This rule, called extended SRv6 rule, allows to use an hash table to associate an incoming packet with its interface, verifying if the packets is coming from a legacy VNF and retrieving the information needed by the *dynamic* SRv6 proxy to process the packet (e.g. re-encapsulating it with the outer IPv6 header and the Segment Routing Header).

CHAPTER 5

Performance Monitoring for Segment Routing over IPv6 based networks

The work I present in this Chapter is mostly taken from my paper *SRv6-PM: a Cloud-Native Architecture for Performance Monitoring of SRv6 Networks*, submitted to *IEEE Transaction on Network and Service Management, 2020* [135].

5.1 Introduction

Novel paradigms such as Software Defined Networking (SDN) [38], Network Function Virtualization (NFV) [121] and Network Virtualization can increase flexibility, reliability of high speed networks when supported by effective tools and systems able to monitor the health of the infrastructure as well as the offered performances continuously and accurately. Classical monitoring tools and protocols have evolved to address the new challenges of softwarized networks, and several new solutions [8] and protocols have been presented, standardized, and effectively applied [163]. In recent years, Segment Routing (SR) networking technology (already covered in Chapter 1, Section 1.1) has come to prominence mainly to deal with the needs of 5G networks or geographically distributed large scale data centers. SR is based on loose source routing whereby a list of *segments* (Segment IDs or SIDs in short) can be included in the packet headers. The segments can represent both topological way-points and specific operations on the packet to be performed in a node. The SR architecture can be deployed on IPv6 networks (SRv6) and, in this case, segments are represented by IPv6 addresses carried in an IPv6 *Extension Header* called *Segment Routing Header* (SRH). According to the SRv6 *Network Programming* concept [27], the list of segments (SID List) can be seen as a "packet processing program", whose operations will

be executed in different network nodes. These operations/functions can be arbitrarily complex, however the most commonly used involve: encapsulation and decapsulation, lookup into a specific routing table, forwarding over a specified output link, Operation and Maintenance (OAM) and Performance Monitoring (PM) functions.

Performance Monitoring (PM) is a fundamental function to be performed in softwarized networks. It allows operators to detect issues in the QoS parameters of active flows that may require immediate actions and to collect information that can be used for the offline optimization of the network. Performance monitoring solutions applied to SRv6 networks can be analyzed by considering several distinct aspects: i) modules, protocols integrated into the data plane to measure and collect data related to nodes and individual traffic flows; ii) modules, protocols integrated into the control plane needed to interact with monitoring entities in the nodes; iii) data management infrastructures specifically designed to store, organize and analyze monitoring data collected in the network.

Regarding the Performance Monitoring data plane subsystem for SRv6, two Internet Drafts have been proposed and are currently under discussion in the IETF SPRING WG. These drafts rely on existing methodologies for performance measurement in general IP and MPLS networks. They propose the extension of such methodologies to PM for SRv6 networks. Both proposed solutions focus on system architecture and protocol specification, but the actual system implementation and integration in the network data plane must still be defined and validated in the field. Moreover, large networks usually consist of hundreds of nodes generating a huge amount of data, and a new class of problem arises when considering the required storage and elaboration capacity. This scenario calls for the integration of a scalable solution with the ability to support common management tasks. Several Network management solutions comprise a cloud ready architecture such as Nagios [117], and therein proposals specifically tackling the requirements of performance monitoring systems can be found, such as [130].

In this Chapter, I describe SRv6-PM which represents an open architecture for Performance Monitoring of SRv6 networks that I heavily contributed to design and implement on Linux-based software routers and switches. Such architecture includes data/-control plane parts and a cloud-native management part based on available open source technologies for supporting Big Data analytics. In order to validate the SRv6-PM architecture, an SDN accurate Per-Flow Packet Loss Measurement (PF-PLM) solution for SRv6 flows has been designed and implemented leveraging the Linux kernel networking stack. The proposed solution integrates the "alternate marking" method described in RFC 8321 [61] and provides an accurate estimation of flow level packet loss (it achieves single packet loss granularity). In addition, the SRv6-PM is equipped with a native cloud architecture that exploits available open source tools specifically designed for collecting topological and time series data related to the various SRv6 traffic flows.

As one of the authors of SRv6-PM, I contributed to both the architecture design and the implementation of several key components of the system. In particular, I designed and implemented several data plane solutions for Per-Flow Packet Loss Measurement and I defined the User API (UAPI) through which an SDN controller is able to manage the flows to be monitored at each node. Therefore, I organized this Chapter to clarify my contributions and how they fit into the overall architecture of SRv6-PM. In Section 5.2, I present the related works and the relevant standards on Performance Monitoring

(PM) as well as the proposed solution for SRv6. In Section 5.3, I discuss the proposed architecture for PM in SRv6 networks and in Section 5.4 the corresponding casting to the PF-PLM solution. In Section 5.5, I describe several Linux kernel-based implementations of the PF-PLM. In Section 5.6, I briefly describe the testbeds used for validating the architecture, while I evaluate the performance of the data plane components in Section 5.7. Finally, I draw some conclusions in Section 5.8.

5.2 Performance Monitoring solutions and standardization

In this Section, I first introduce some general definitions and classification of performance metrics used in Softwarized Networks. Then, I provide an overview on the related works on Performance Monitoring as well as on related standards for IP and MPLS networks. Finally, I discuss specific solutions for SRv6 proposed for IETF standardization.

5.2.1 Performance Metrics and Measurement Methods

Several parameters need to be measured for detecting performance degradation or outage in telecommunication networks and there are different methods to measure. The most commonly considered metrics at network level are: i) packet delay (also known as latency); ii) packet loss; iii) throughput. *Delay* is the time required to transmit a packet across a link or a network path and can be measured as either one-way[1] or as round-trip delay[2].*Packet loss* occurs when one or more packets fail to reach the destination either because of transmission errors or because of network congestion. It is usually measured as the percentage of lost packets with respect to sent ones. The *throughput* can be measured in bit/s or packet/s and corresponds to the rate of (successful) information transfer over a link or a network path.

Performance measurement methods can be broadly classified into passive, active or hybrid (a combination of active and passive) methods [115]. Passive ones are based on the analysis of the traffic passing through the network. Usually dedicated devices called sniffers are adopted to analyze the traffic flows and acquire statistics that can be provided to the monitoring systems. Passive approaches have the advantage of not increasing the traffic on the network, but unfortunately they are often not very effective in spotting when and where there are anomalies in the network. Conversely, active methods generate streams into the network to measure the performance of a path with the goal of monitoring the interesting metrics from source to destination. The side effect of this approach is that active methods increase the traffic in the network.

5.2.2 Performance Monitoring in Softwarized Networks

There are many commercial and open source solutions for network performance monitoring. Some of them rely on generic tools that, among other features, include the monitoring of network devices. Two notable examples are Nagios [117] and Zabbix [186]. Other solutions are based on tools developed to monitor cloud environments, such as Ceilometer [30] for example, which is adopted in OpenStack deployments. In the SDN, OpenFlow protocol is the industry standard de facto, and its interface allows

[1]The time spent from the source to the destination.

[2]The one-way latency from source to destination plus and the one-way latency from the destination back to the source.

controllers to obtain from nodes numerous statistics about the flows that the device is managing [122]. Several studies have proposed solutions for OpenFlow networks. For example in [168] OpenNetMon proposes a framework to measure throughput, delay, and packet loss of traffic flows in OpenFlow networks. The work focuses on how to collect data effectively to provide controllers with a network-wide vision, but limiting the additional computational load needed to obtain all measurements. Another solution for OpenFlow is proposed in [141] where the cost of such measures is also discussed. A recent overview of the activities on Openflow traffic monitoring is provided in [130]. A general review of other monitoring techniques and solutions for softwarized networks can be found in [163].

5.2.3 Active Monitoring in IP and MPLS networks

Active measurements 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". Several research works and standards have been proposed both for IP and MPLS networks, especially in the IETF framework. Among them, RFC 4656 "The One-Way Active Measurement Protocol (OWAMP)" [140] provides capabilities for the measurement of one-way performance metrics in IP networks, such as one-way packet delay and one-way packet loss. RFC 5357 "Two-Way Active Measurement Protocol (TWAMP)" [78] introduces the capabilities for the measurements of two-way (i.e. round-trip) metrics. These specifications describe both the *test protocol*, i.e. the format of the packets that are needed to collect and carry the measurement data and the *control protocol* that can be used to set up test sessions and to retrieve measurement data. For example, OWAMP defines two protocols: "OWAMP-Test is used to exchange test packets between two measurement nodes", and "OWAMP-Control is used to initiate, start, and stop test sessions and to fetch their results" (quoting [140]). Note that in general there can be different ways to set up a test session: the same test protocol can be re-used with different control mechanisms.

RFC 6374 [37] specifies protocol mechanisms to enable the efficient and accurate measurement of performance metrics in MPLS networks. The protocols are called LM (Loss Measurement) and DM (Delay Measurement). I will refer to this solution as MPLS-PLDM (Packet Loss and Delay Measurements). In addition to loss and delay, MPLS-PLDM also considers how to measure throughput and delay variation with the LM and DM protocols. Differently from OWAMP/TWAMP, RFC 6374 does not rely on IP and TCP, and its protocols are streamlined for hardware processing. While OWAMP and TWAMP support the timestamp format of the Network Time Protocol (NTP) [39], MPLS-PLDM adds support for the timestamp format used in the IEEE 1588 Precision Time Protocol (PTP) [72]. There are several types of channels in MPLS networks over which loss and delay measurement may be conducted. Normally, PLDM query and response messages are sent over the MPLS Generic Associated Channel (G-ACh), which is described in detail in RFC 5586. RFC 7876 [142] complements the RFC 6374 by describing how to send the PLDM response messages back to the querier node over UDP/IP instead of using the MPLS Generic Associated Channel.

5.2.4 SRv6 Performance Monitoring

The standardization activity regarding the Performance Monitoring of SRv6 networks is very active, and two internet drafts have been published so far:

i Performance Measurement Using UDP Path for Segment Routing Networks [59]

ii Performance Measurement Using TWAMP Light for Segment Routing Networks [58]

Both solutions provide the possibility of measuring delay and loss of a single SRv6 flow, characterized by a *SID List*. It is worth noting that an SRv6 flow can be identified in other ways such as IP Source Address, Destination Address, Flow Label as discussed in [27].

The data collection takes place with test UDP packets transmitted on the same measured path. Test UDP packets collect the one-way or two-way PM data and make them available to the node which requested the measurement.

The [59] aims at extending and reusing the MPLS-PLDM work defined in RFC 6374 [37] and RFC 7876 [142]. It specifies procedures for using a UDP path for sending in-band probe query and response messages for Delay and Loss performance measurement. Although the RFC 6374 applies only to MPLS Networks, the specified procedures are applicable to SR-MPLS and SRv6 data planes for both links and end-to-end measurement for SR Policies. The draft introduces TLV (Type-Length-Value) coding to specify the Return Path and a TLV for traffic coloring. Another interesting characteristic of the proposal is the possibility to send the probe response directly to an external controller.

The most active standardization path [58] promotes the adoption of a modified version of the TWAMP Light protocol defined in RFC 5357 Appendix I and its simplified extension Simple Two-way Active Measurement Protocol (STAMP) proposed for standardization in [112]. These protocols lack support for Loss Measurement in traffic flows that are required in SRv6 networks. Thus, the draft considers procedures and messages defined in RFC 5357 for Delay Measurement (DM) and specifies new procedures and messages for Loss Measurement both for SR-MPLS and SRv6 data planes. I select this draft as reference procedures for the performance monitoring architecture presented in this Chapter and implemented for evaluation.

Moreover, both the solutions support the Alternate-Marking Method defined in RFC 8321 [61] for accurate loss monitoring and that can be applied to IPv6 and SRv6 flows as specified in [56] and presented by [113]. Indeed the presence of in flight packets makes it difficult to obtain an accurate evaluation of the number of lost packets, as discussed in RFC 8321 [61]. The proposed solution combines packet “marking” and packet “counting” to cope with this problem. I integrated this technique in the accurate loss monitoring solution, and a detailed description is provided in Section 5.4.

5.3 Monitoring architecture

5.3.1 Performance Monitoring: Data and Control Planes

To monitor the QoS experienced by the SRv6 traffic flows, the controller needs to interact with the network routers and switches. The main operations are to start/stop the

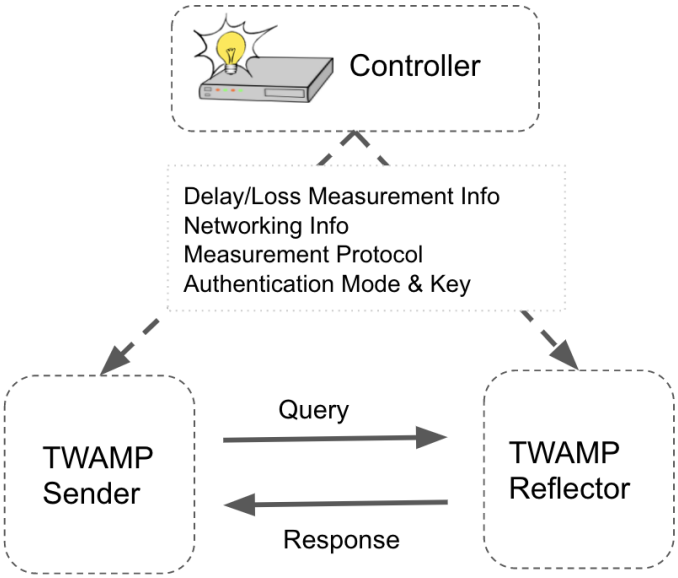


Figure 5.1: *Architecture of Performance Monitoring Data and Control planes.*

monitoring procedures on the selected flows, and then to collect the measurement data such as packet loss ratio and delay. Moreover, a data plane measurement protocol is needed among the monitored nodes. In this regard, the approach described in [58] was adopted in SRv6-PM. It uses and extends the TWAMP Light protocol which appears to be more suitable for the IPv6 data plane with respect to [59] (conceived for an MPLS data plane). Figure 5.1 shows the reference architecture for the data plane monitoring protocol, including the interaction of the data plane protocol entities with the controller.

Two entities are considered, named *TWAMP Sender* and *TWAMP Reflector* respectively. The *TWAMP Sender* sends a probe *Query* to the *TWAMP Reflector*, which replies with a probe *Response* message. The Query and the Response messages carry the performance monitoring information. The controller, shown in Figure 5.1, is used to provision the *TWAMP Sender* configured accordingly to carry out the measurements to start/stop the experiment and to collect the results. It is worth noting that the presence of the controller avoids the need for a "control plane" monitoring protocol between the *TWAMP Sender* and the *TWAMP Receiver*. In fact, all control and management operations are performed by the controller.

Both the probe query and response messages are sent on the congruent path of the data traffic by the sender node. They are then used to measure the delay of an SRv6 traffic flow or to collect counters related to a specific flow. The controller needs to specify the type of measure that needs to be performed, the SID List of the monitored flow, the relevant networking information such as the UDP port and the destination address, the authentication mode and keys.

The controller interacts with the network nodes using an API, which needs to be carefully designed. In this regard, the SRv6 Southbound API proposed in [172] has been extended by introducing additional methods for gaining full control over the SRv6 data plane and the routing of a Linux node, as well as for handing messages related to the Loss Monitoring. In SRv6-PM, the SRv6 Southbound API is implemented using

gRPC which provides methods for executing and controlling the measurement sessions. Moreover, since the monitoring architecture proposed in [58] does not provide any data collection system, the Southbound API has been extended to include: i) a mechanism operated by the Controller to collect data on nodes; ii) a mechanism to enable the Controller for pushing the Measurement data to the Cloud Native Data infrastructure implemented in SRv6-PM. More details about the SRv6 Southbound API are available in [135].

5.3.2 Cloud Native Big Data Management

Data retrieved from measurement processes should be collected along with a rich set of companion data describing the state of the network at that specific time, possibly including other contextual information. Therefore, such data can be exploited for two main goals: i) real-time monitoring; ii) offloading analysis and optimization. In the first case, data should be processed in real-time fashion to react properly to sudden failures, changes of traffic patterns and so on. In the second case, potentially huge sets of historical data can be processed to provide long term insights using the more suitable and powerful tools. Considering that the amount of information to be collected and processed in a realistically sized IP backbone is massive, a dedicated Big Data infrastructure seems to be the only solution to meet the two goals. For this reason, the SRv6-PM architecture envisages the use of a Big Data infrastructure to elaborate the collected performance data through a multi-stage pipeline processing. The processing stages involved range from the *Data provisioning* to the *Data visualization*. [135] provides a thorough description of the Big data infrastructure, the open source tools as well as the fully integrated Continuous Integration/Continuous Development (CI/CD) environment aiming to simplify the deployment of the whole processing pipeline.

5.4 SRv6 Accurate Loss Measurement

To validate the SRv6-PM architecture, a full Per Flow Packet Loss Measurement (PF-PLM) solution was designed and implemented. PF-PLM is compliant with the draft [58], adopts the step detection and the alternate marking technique for counting described in [113]. I consider the reference network scenario depicted in Figure 5.2 comprising an SRv6 network domain. IP traffic arrives at an ingress edge node of the network, where it can be classified and encapsulated in an SRv6 flow. 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 Segment Routing Header (SRH) of the outer IPv6 packet. The outer IPv6 packet crosses the network (according to its *SID List*) and arrives at the egress edge node where the inner IP packet is decapsulated (the outer IPv6 and SRH are removed). For example in the Figure 5.2, node A acts as the ingress node while node B as the egress node for the green packets. The ingress node A applies the SR policy, i.e. it writes the *SID List* into the SRH header.

5.4.1 Packet Counting

To perform loss measurement, it is necessary to implement packet counters associated with SRv6 flows, both in the Ingress node and in the Egress one. For this purposes, an SRv6 flow corresponds to an SR policy, i.e. to a *SID List* and the counting process for a

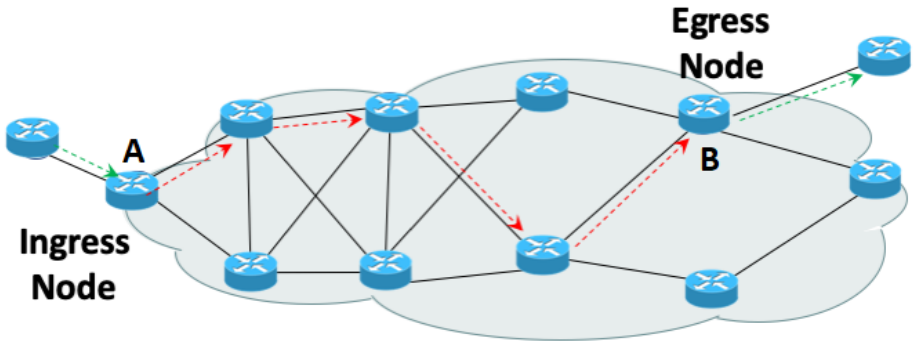


Figure 5.2: *Reference SRv6 network scenario for Performance Monitoring.*

set of flows (identified by their *SID Lists*) can be turned on/off explicitly. In an Ingress node, this means processing all outgoing SRv6 packets and counting the packets that belong to the set of monitored flows (by comparing the *SID List* of outgoing packets with the *SID Lists* of monitored flows). Similarly, in an Egress node, this involves processing all incoming SRv6 packets, checking whether the packets belong to the set of monitored flows, and incrementing the counters accordingly.

These counting operations can have a high processing cost for a software router. For this reason, I gave special attention to their design and implementation within the Linux kernel networking stack. (see Section 5.7 to evaluate their impact on the processing performance).

5.4.2 Traffic Coloring

Comparing the transmission counter with the receiver one is a straightforward operation to be implemented, but it requires that the two counters refer to exactly the same set of packets. However, since flows cannot be stopped it is difficult to get an accurate loss evaluation. Indeed, to achieve high accuracy in detecting single packet loss events while a flow is active, one needs to properly consider "in-flight" packets, i.e. packets counted by the Ingress node but not yet counted by the Egress one.

The solution proposed in the RFC 8321 [61] is to virtually split packets into temporal countable blocks and by "coloring" the packets of the flows to be monitored with at least two different marks so that different consecutive blocks will have different colors. For example, a continuous block of packets of a flow is colored with color R (e.g. for a configurable duration T), then the following block of packets is colored with color B (again for a duration T), and so on. In this case, two separate counters are needed both in the Ingress node and in the Egress node for packets with color R and with color B. In general, this solution requires for each flow a single counter per color.

To evaluate the packet loss, one can read the inactive counters. In the previous example, when the active counter is B, it is possible to safely read counters for color R from the Ingress and Egress nodes and evaluate their difference, which exactly corresponds to the number of lost packet in the previous interval (see Figure 5.3). Usually, it is worth waiting some time to read the counters after the color switch in order to give time to in-flight packets to arrive at the Egress node. The PF-PLM solution has

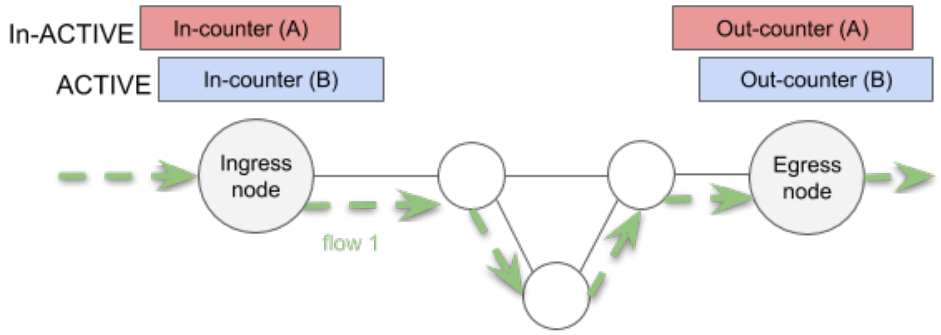


Figure 5.3: *Alternate coloring method (RFC 8321).*

been designed to wait for $T/2$ before reading the inactive counter. However, the main drawbacks of this technique are the large number of counters needed and the limited measurement frequency.

When the alternate marking method is applied on a given network, the mechanism for coloring packets must be specified. In [56], the authors address several marking solutions involving IPv6 Extension Headers, IPv6 addresses, or Flow Labels but they conclude that none of these marking methods could be safe enough to be standardized. Since the PF-PLM solution is tailored for SRv6 networks, the intrinsic network programming model could be used for realizing different marking solutions. During the design of PF-PLM, I considered two different alternatives: i) modification of the DS field, previously known as IP Type of Service (TOS) field; ii) encoding the color in a SID of the *SID List* present in the SRH.

The i) solution is simple but has some drawbacks: the number of bits available in the DS field is limited (6 or 8) and they are considered precious. Only one bit is required to encode two colors. In addition, an extra bit can be used to differentiate the colored traffic to be monitored from the non-colored traffic not to be monitored. This can be useful to avoid comparing the full *SID List* to decide whether a packet is part of a flow under monitoring or not. In the implementation described in Section 5.5, I have used two bits of the DS field.

The ii) solution encodes the color in a SID according to [27] where an IPv6 address representing a SID divided in LOCATOR:FUNC:ARGS. The LOCATOR part is routable and allows the forwarding of the packet towards the node where the SID needs to be executed. The FUNC and ARG parts are processed in the node that executes the SID. In particular, the ARG part can reserve a number of bits for the alternate marking procedures. This may accommodate the use of more than two colors. This solution, however, has an implementation drawback: due to the variable position of these bits, implementing a hardware processing solution is much harder, and can be out of reach for current chips that need to operate at line speed. Moreover, periodically changing the ARG bits in a SID of a running flow can cause an interference with the SRv6 forwarding plane (e.g. for Equal Cost MultiPath) when the SID is used as IPv6 destination addresses.

The PF-PLM solution relies on the TWAMP Light protocol which has been originally defined as an active monitoring technique. On the other hand, PF-PLM exploits a combination of the Alternate Marking method ([61]) and local counters, that is usually

considered as "hybrid" monitoring since no packets are added but only modified (i.e. colored). As discussed in RFC 8321, if the packets to be monitored natively include some bits that can be used for marking, the PF-PLM solution becomes "passive" with no need of substantial modification.

5.4.3 Data Collection

The data collection is carried out by the TWAMP Light protocol extension specified in [58]. The Sender prepares a query message that is sent in the congruent path with traffic, and the Reflector replies with a response message that can be sent in-band, in the reverse path of the data traffic or out-band.

TWAMP messages are inserted into an UDP/IPv6 packet. They are sent with Source and Destination UDP ports configured by the user. However, the UDP destination port is used to identify the message type and the authentication mode, and since TWAMP does not have any indication to distinguish between query and response messages, the source port needs to be different from the destination port.

The UDP packet is then encapsulated in another IPv6 packet comprising the SRH as for normal data traffic. To distinguish such as packet from normal traffic, a special policy is applied using the Endpoint function $END.OP$ as defined in [9] that modify the target SID to punt the packets in the Egress node. In the following, I provide a brief description of the simple authenticated version of TWAMP messages with the relative fields to better clarify the data collection procedure.

TWAMP Query message

In Figure 5.4 a) there is a depiction of the query message that includes a Sender Sequence Number (32-bit), the Sender Transmit Counter (64-bit) and the Block Number (8-bit) for the specific flow that is going to be monitor. The Block Number identifies the considered color and it should refer to the inactive one. Flags are used to specify some options, such as the counter format and the type counting mode (bytes or packets). Finally, the control code (Ctrl Cod) indicates to the Reflector if the response is required in-band or out-band.

TWAMP Response message

The response message includes data collected in the Reflector and data received from the sender, i.e. the Sender Sequence Number, the Transmission Counter and Block Number. The Reflector includes the Reception counter for the flow specified in the SR Header and if the response is in-band, the Sequence Number, the Transmission Counter and Block Number for the return path.

When the sender receives the response, it is able to calculate the loss in the forward path using the Sender TX Counter and Reflector RX Counter. It can also calculate the loss in the return path using the Reflector TX Counter and reading locally the corresponding RX counter.

5.5 Monitoring System Implementation

The Performance Monitoring system has been implemented using several open source platforms and software frameworks. The SRv6 Manager, defined in [172], was im-

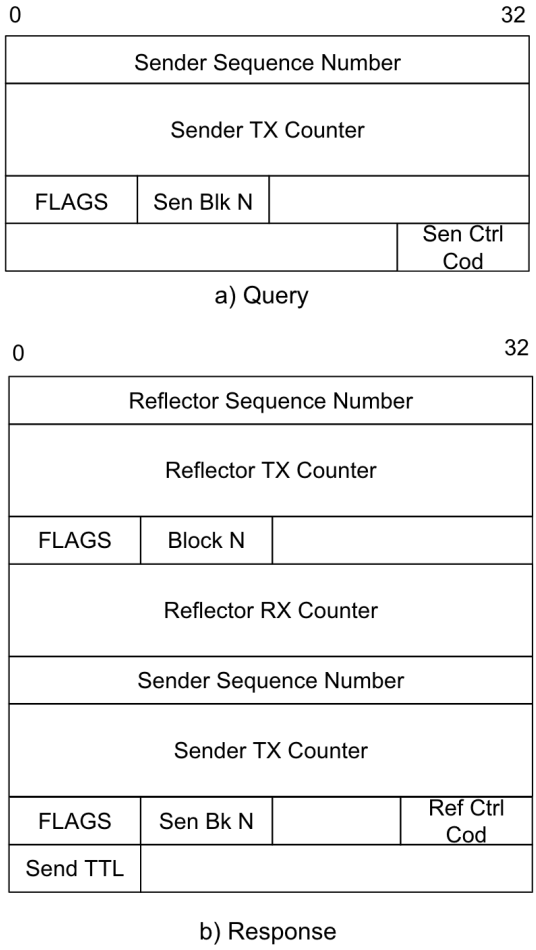


Figure 5.4: *TWAMP Loss Measurement a) Query and b) Response messages defined in [58].*

proved for controlling the *TWAMP Sender* and *Reflector*. Specifically, it provides the Southbound loss monitoring interface which allows the SDN Controller to communicate with the nodes in order to configure, start and stop the measurements. Regarding the data plane side, I extended the Linux networking stack capabilities by implementing the PF-PLM solution through different in-kernel packet processing frameworks, namely *Iptables*, *Ip set* and *extended Berkeley Packet Filter (eBPF)*. Both *Iptables* and *IP set* approaches are built upon the well-known Netfilter framework which enables various network-related operations to be implemented under the form of customized handlers. Although Netfilter-related approaches turn out to be very flexible and popular, they certainly do not stand out for the impressive attainable performance. On the other hand, *eBPF*-based solutions exploit an in-kernel Virtual Machine (see Chapter 2, Section 2.3) that provides very efficient packet processing at the price of writing, strictly following a set of (not always obvious) constraints, specific network programs.

In the following Subsections, I am going to provide an overview on the systems and tools that I have designed and implemented/extended, as well as detailed descriptions of my contributions. All the developed software components are available as open source [158].

5.5.1 Linux SRv6 subsystem

The Linux kernel SRv6 subsystem supports the basic SRv6 operation described in [25] and most of the operations defined in [27]. A Linux node can classify incoming packets and apply SRv6 policies, i.e. encapsulate the packets with an outer packet carrying the list of SRv6 segments (SIDs). A Linux node can associate a SID to one of the supported operations, so that the operation will be executed on received packets that have such SID as IPv6 Destination Address. An in-depth description about the Linux SRv6 implementation with a list of the currently supported operations can be found in Chapter 2.

5.5.2 Linux Netfilter/Xtables/Iptables subsystem

The Netfilter/Xtables/Iptables (see Chapter 2) allows the system administrator to insert chains of rules for the processing of packets inside predefined tables (`raw`, `mangle`, `filter`, `nat`). Each table is associated with different types of packet processing operations. In each chain, packets are processed by sequentially evaluating the rules in the chains.

As shown in Figure 5.5, there are five processing stages (or phases) (`PREROUTING`, `INPUT`, `FORWARD`, `OUTPUT`, `POSTROUTING`) where the default chains associated with specific tables are processed. Moreover, it is possible to create additional chains as needed. For example, in the `POSTROUTING` stage, the default post-routing chains associated to the `mangle` and `nat` tables are processed. *Iptables* is a generic firewalling software whose user space CLI (Command Line Interface) utilities allow a system administrator to configure the tables/chains/rules provided by the Netfilter/Xtables subsystem. Figure 5.5 shows two additional chains (`BLUE-CHAIN`, `RED-CHAIN`) that I introduced in the packet loss monitoring solution, visible in the rightmost part of the figure. For the sake of clarity, I will use *Iptables* to refer in general to the whole Netfilter/Xtables/Iptables architecture and subsystem, including the IPv6 specific modules.

Iptables is highly modular and can be extended. In particular, it is possible to develop a custom packet matching module, to specify a rule which refers to the custom module name and includes extra commands depending on the specific extension. I have followed this approach, as described in Subsection 5.5.2.

Iptables based PF-PLM implementation

In order to evaluate Per-Flow Packet Loss metrics, it is necessary to collect information on both Ingress and Egress nodes in a coordinated way.

In the Ingress node, the traffic is classified (based on IP Destination Addresses) and can be associated with an SRv6 policy (step 1 in Figure 5.5). This means that a matching packet can be encapsulated in an outer IPv6 packet with a new IPv6 header followed by the Segment Routing Header with the proper Segment List (SID List) (step 2 in Figure 5.5). The encapsulated packet continues its journey in the networking stack until it reaches the `POSTROUTING` chain of the *mangle* table (step 3 in Figure 5.5). Right here, I add a "jump" rule with the aim to divert all SRv6 traffic to a custom chain for statistic collection purposes (step 4 in Figure 5.5). In particular, I have two custom chains, named *coloring chains*, as I need one color for each chain to implement the "alternate marking" approach. Each chain contains a set of rules, each rule is used to

match on a specific SID List included in the SRH header. At any given time, only one of the two chains is *active* and is referred to by the "jump" rule.

When an SRv6 packet enters the active *coloring chain*, rules contained in the chain are applied to the packet in sequence until:

1 there is a rule that matches the SID List of the packet. The match counter of this rule is incremented. The packets need to be colored with the color corresponding to the active *coloring chain*. When the coloring based on the DS field is used, the *color bit* of the DS field will be set to the proper color (0 or 1). Once this is done, the packet processing comes back to the calling chain, i.e. the `POSTROUTING` chain in the `mangle` table;

2 all rules have been considered and no match with SID List of the packet has been found, so the packet processing simply jumps back to the calling `POSTROUTING` chain.

In the Egress node, packets belonging to the flows to be monitored enter from an ingress interface and arrive encapsulated in an IPv6 outer header. Therefore, it is required to match on the SID List of incoming packets. Hence, I use the `PREROUTING` chain of the `mangle` table to match on the different colors, considering the color bit of the DS field (step 1 of Figure 5.6). Then for each color, I process the packet in a chain which includes one rule for each SID List that has to be monitored (step 2 of Figure 5.6). I refer to this custom chain as *color-counting chain*. When a rule matches the SID List in the packet, the match counter of the rule is incremented. After the processing in the *color-counting chain*, the normal packet processing continues. Since packets are handled at the egress node, the destination address will be a SID that corresponds to a packet decapsulation operation. This SID is matched in the routing operation (step 3 of Figure 5.6), then the decapuslation operation is executed by the SRv6 Linux kernel subsystem (`encap seg6local`, step 4 of Figure 5.6).

Improvements to the SRH match extension

The Netfilter/Xtables already provides a module extension that matches packets based on the Segment Routing header of a packet. The module is named `ip6t_srh` and it has been included in the Linux kernel tree since version 4.16. Through the `ip6t_srh` extension, it is possible to write Netfilter/Xtables rules that match the current SID, the next SID and the number of the left segments in a SRH. However, it is not possible to match packets by comparing a full SID List. For this reason, I extended `ip6t_srh` to support matching on the whole SID List.

The improvements to `ip6t_srh` affect both the kernel module and the user space library designed to interact with it. On the kernel side, I patched the module in order to store the SID List provided by the user space while creating an *Iptables* rule. Furthermore, I also implemented a match function that compares this SID List with the one present in the examined SRv6 packet. When the two SID Lists are identical, the match function returns successfully and the kernel proceeds with the execution of further match extension modules present in the rule and/or with the execution of a target action. If the two lists differ, the match function returns an error which allows Netfilter/Xtables to move onto the next rule, if any.

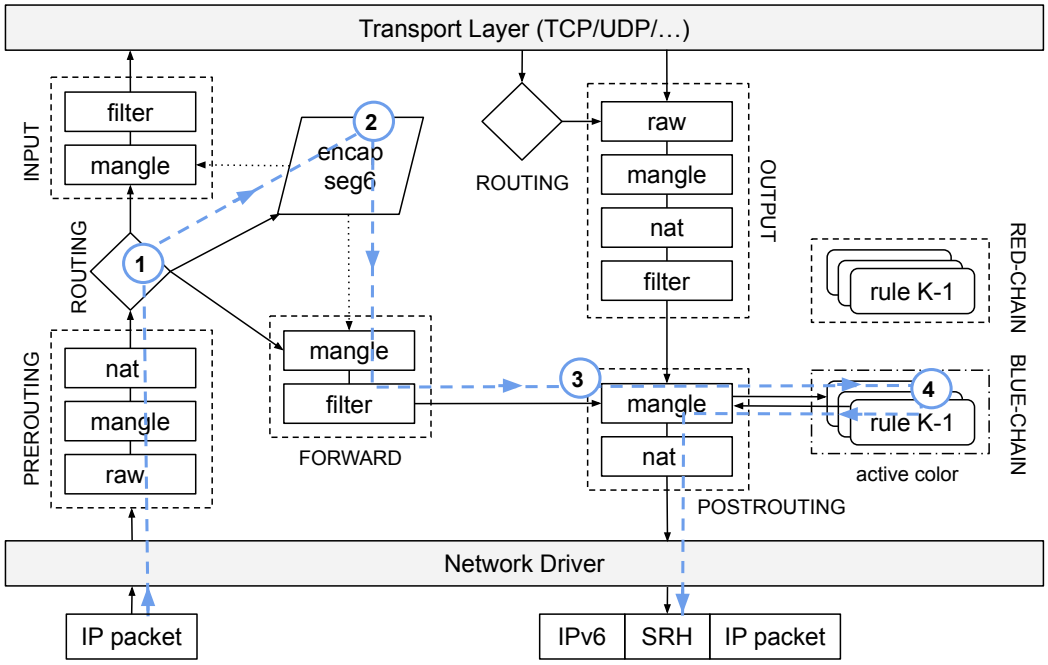


Figure 5.5: *Netfilter/Xtables based packet processing in the Ingress node.*

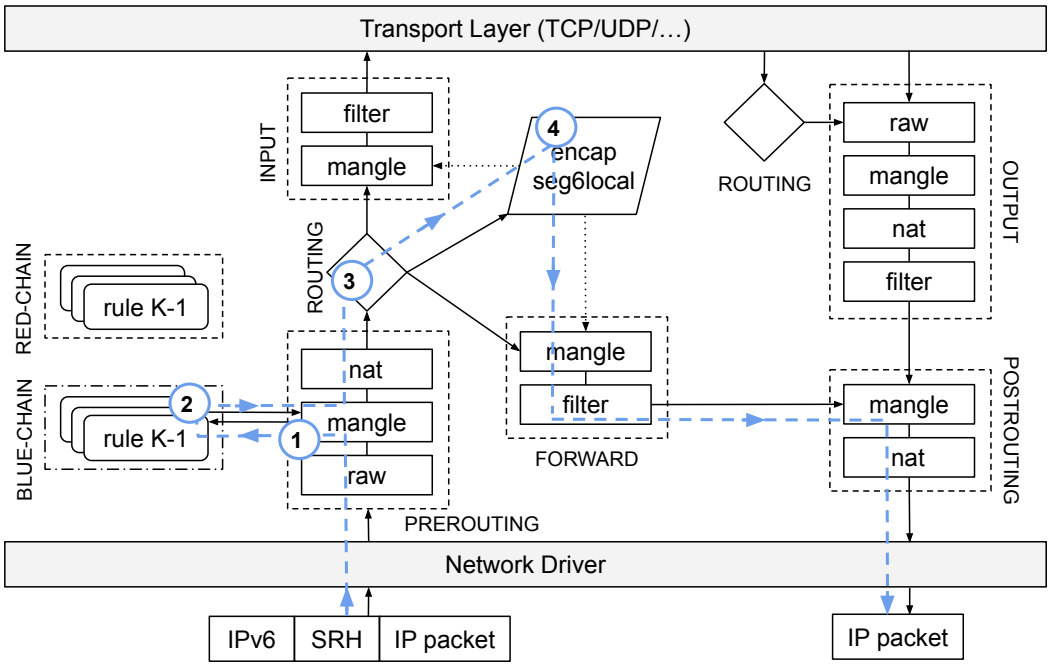


Figure 5.6: *Netfilter/Xtables based packet processing in the Egress node.*

For what concerns the user space side, I updated the `lib6t_srh.c`. This library is used to parse the commands provided by the user on the CLI and then to create, initialize and populate the data structures required to build a rule. The same data structures are used by `ip6t_srh` module to carry out matches on SRv6 packets. Therefore, I added the parsing function that reads and validates a list of SIDs supplied by the user and the fields needed to store the validated SID List. I have also implemented the helper

functions to retrieve and display the SID List and the number of matched packets for each rule. The last feature is fundamental to count the number of packets belonging to a given flow in a SRv6 network.

In Listing 5.1, I show how easy is to define a new *coloring chain* for processing SRv6 traffic and to monitor a new flow whose SID List is equal to `sid10, sid1`.

Listing 5.1: *Iptables userspace tool used for configuring a new SRv6 flow to be monitored.*

```
1) ip6tables -A POSTROUTING -t mangle   \
     -m rt --rt-type 4 -j blue-chain

2) ip6tables -A blue-chain -t mangle    \
    -m srh --srh-sid-list sid0,sid1     \
    -j TOS --set-tos 0x01/0x01
```

The command (1) creates a new chain aiming to process only SRv6 traffic. Conversely, command (2) adds a rule to the `blue-chain` of the `mangle` table and asks Netfilter/Xtables framework to use the `srh` module to match an SRv6 packet considering the SID List indicated by my new the attribute `--srh-sid-list`. The name I chose for the new attribute is simple, easy to understand, and allows a user to specify the list of SIDs by simply separating them with a comma. In this example, the action (`-j --set-tos 0x01/0x01`) specified in the target consists in marking the packet by setting the first bit of the DS field of the outer IPv6 header.

Note that with the new `--srh-sid-list` attribute, a user can associate a flow with any legal action (via the appropriate target) needed for purposes other than just performance monitoring, i.e.: traffic accounting, quotas, shaping, and so on.

Performance issues of *Iptables* based PF-PLM

Rules which are included in the *coloring chains* (Ingress node) and in the *color-counting chains* (Egress node) are tested sequentially, until a matching rule is found (or all the rules have been tested).

This sequential scan approach can have a significant impact on the node throughput due to the processing load for comparing the *SID List* in the SRH packets with the *SID Lists* in the *coloring chains* or in the *color-counting chains*. On average, the processing load increases linearly with the number of rules. This number of rules corresponds to the number of flows for which it is desired to monitor packet loss. In the performance experiments, I will analyze the impact of the number of monitored flows on the node throughput due to the sequential scan approach.

5.5.3 The *IPset* framework

IP set [73] is a framework inside the Linux kernel, provided through Xtables-addons [184], which extends the Netfilter/Xtables/Iptables capabilities. *IP set* allows a user to create rules that match entire *sets* of elements at once. In contrast to normal *Iptables* chains which are stored and traversed linearly, elements of *sets* are stored in indexed data structures for very efficient lookup operations, even when dealing with very large sets.

Depending on the needs, an *IP set* can store IP host addresses, IP network addresses, TCP/UDP port numbers, MAC addresses, interface names, or a combination of them to ensure outstanding performance when matching an entry against a set. To use *IP set*, uniquely named sets are created and populated using the command line `ipset`. These sets are then referenced in the *match specification* of one or more *Iptables* rules. Hence, the `iptables` command refers the set with the match specification `-m set --set foo dst`, which means "match packets whose destination is contained in the set with the name *foo*". As a result, a single `iptables` command is required regardless of the number of elements in the *foo* set. To achieve the same result using only *Iptables*, it would be necessary to create a chain and insert as many rules as the elements contained in the *foo* set.

IP set provides different types of data structures to store the elements (addresses, networks, etc). Each set type has its own rules for the type, range and distribution of values it can contain. Different set types also use different types of indexes and are optimized for different scenarios. The best/most efficient set type depends on the situation. The hash sets offer excellent performance in terms of speed of the execution time of lookup/match operation and they fit perfectly to my needs. Assuming to insert N IP addresses in the hash set *foo*, the cost of searching for an address is asymptotically equal to $O(1)$. Conversely, the same operation with *Iptables* would have a cost of $O(N)$.

The *IP set* framework is designed to be extensible: there are several header files containing helper functions and templates that, through very clever use of C macros and callbacks, allow developers to define new *sets* and tailor the code to fit their own needs.

IP set based PF-PLM implementation

With the *Iptables* based PF-PLM implementation, packet processing throughput decreases when the number of flows to be monitored increases. Note that not only the throughput of the monitored SRv6 traffic is affected, but also the traffic that does not need to be monitored. To overcome this shortcoming, I have designed and implemented an enhanced solution based on *IP set*.

The *IP set* framework does not natively allow storing elements of `SID List` type within a hash set. Therefore, in order to exploit the *IP set* for my purposes I patched the framework by creating a new brand hash set called `sr6hash`. Through this improvement, I can store *SID Lists* related to flows to be monitored in *sets* and perform efficient lookup operations on those collections.

To support the new `sr6hash` hash type, I patched the *IP set* framework on both the user space and the kernel space sides. On the user space side, I defined a new data structure, the `nf_srh`, which contains the SRH with a *SID List* whose maximum length is fixed and set at compilation time (16 SIDs in my experiments). Moreover, I added two new functions to the *IP set* framework libraries:

- `ipset_parse_srh()` is designed for parsing the *SID List* supplied by the user through the *IP set* CLI and for creating and populating the `nf_srh` data structures;
- `ipset_print_srh()` is designed for displaying all the *SID Lists* in a given

`sr6hash` (hash set) set along with their match counters.

In Listing 5.2, I report the patched user space commands used to: 1) create a hash set of type `sr6hash`; 2) add a *SID List* consisting of two SIDs; 3) add an *Iptables* rule, in which the match is performed on the `blue-ht` hash set and the action consists in marking the first bit of the DS (tos) field in the outer IPv6 packet.

Listing 5.2: *IPset userspace tool extended for supporting the new `sr6hash` (hash) set.*

```
1) ipset -N blue-ht sr6hash counters

2) ipset -A blue-ht                        \
     2001:db8::1,2001::db8::2

3) ip6tables -A blue-chain -t mangle       \
     -m set --match-set blue-ht dst        \
     -j TOS --set-tos 0x01/0x01
```

On the kernel side, I introduced a new *IP set* module which is the implementation of the hash set `sr6hash`. In particular, I have defined:

- the data structure of an element of the hash set (`struct hash_sr6_elem`) containing the SRH with the *SID List* to be stored. The user space and kernel space data structures are identical so that it facilitates the exchange of information between the two contexts;
- the equality function `hash_sr6_data_equal()` to compare two *SID Lists*;
- the functions `hash_sr6_kadt()` and `hash_sr6_uadt()` which are used for adding, deleting an element to/from the hash set and testing the membership. Two different functions are needed since they are called from different contexts, respectively the kernel space context and the user space one;
- the `ip_set_type` structure where I set the properties, policies and extensions supported by the `sr6hash` module. Therefore, this structure is used as a "glue" that sticks parts together and is used to, actually, register the hash set when the module is loaded into the kernel and to deregister it when it is unloaded.

5.5.4 The eBPF based counters implementation

The Loss Monitoring implementations presented so far are based on Linux kernel modules which carry out all the needed operations for parsing packets, updating flow counters and coloring. Kernel modules allowed me to implement the counting system in a very effective way, but this approach comes with a number of disadvantages that I briefly report hereafter. Every time I need to update the Linux kernel, I also have to maintain the counting modules updated. Furthermore, if I need to add a functionality to my counting implementation, I have to remove the loaded modules and to replace them with updated ones. Replacing a module is not an atomic operation: the time between the removal and the loading of the updated module causes a down-time of the monitoring service, which may not be tolerated if it is necessary to account statistics

on high-rate data flows. A kernel module can have unconditional access[3] to most of the internals of the system, and if there is a bug, this can compromise the overall system security and stability. In addition, the *Iptables* and *IP set* modules are hooked to the Netfilter, which can not guarantee the maximum performance in terms of packet processing.

An eBPF network program, despite its intrinsic limitations, can solve the issues that arise by implementing the counter using kernel modules. Therefore, I designed two eBPF programs attached to specific networking hooks with the purpose of entirely replacing the previous implementations.

Both `tc_srv6_pfplm_egress` and `xdp_srv6_pfplm_ingress` eBPF programs are written in C and compiled through the Clang/LLVM compiler [128]. The result is a single object file called `srv6_pfplm_kern.o` where each program is placed in a specific text section so that it can be loaded independently from the other. The two eBPF programs are designed to account individually the ingress and egress flows that match some given SR policies, also taking into account the color marking.

Every monitored flow is stored in eBPF maps whose purpose is to share the information among the two eBPF programs and the user space. Flow lookups and match operations are fast and efficient thanks to the `BPF_MAP_TYPE_PERCPU_HASH` map type. This type of map allows flow counters to be retrieved using the *SID List* as key in a lock-free fashion and in constant time, regardless of the number of SR policies present in the map. Currently, all the hashtable data structures provided by the eBPF infrastructure support fixed length keys. Thus, to maximize the lookup performance and minimize memory waste, I created N maps/hashtables, and in each map I store all the flows with the same *SID list* length. The value of N can be decided at compile time, and in my current implementation the default max *SID List* length is equal to $N = 16$ SIDs. Therefore I have come up with a total of 16 maps/hashtables. Furthermore, I have also decided to store and keep separated the ingress flows to be monitored from the egress ones, hence the total number of flow maps/hashtables is doubled.

I keep all the necessary maps persistent and accessible as files placed in the */sys/fs/bpf/pfplm* directory and to interact with them I provided a User API (UAPI) released in the form of a shared library. Such UAPI hides the complexity of the underlying data structures and allows the user to change the active color for marking packets, to easily add/remove flows in ingress/egress directions and to access flow statistics.

In Figure 5.7, I illustrate the processing and management of statistics in the Egress node realized with the eBPF program named `xdp_srv6_pfplm_ingress`. In the Egress node, IPv6 packets are analyzed to find out if there are SRH containing *SID Lists* related to flows to be monitored. In this case, the PF-PLM packet processing is carried out directly in the driver space, leveraging the ingress eBPF eXpress Data Path (XDP) hook. XDP is able to intercept RX packets directly at the NIC driver and (possibly) before the socket buffer (`sk_buff`) gets allocated for obtaining the highest performance. For more details about the eXpress Data Path (XDP) and how it is related to the Linux kernel networking stack, please refer to Chapter 2, Section 2.1. Specifically, the *SID List* of each IPv6 Segment Routing packet is extracted and based on the length of such *SID List*, the suitable flow map is retrieved. At this point, the lookup of the *SID List* is performed and, if the flow is found, the per-flow counters are

[3] A kernel module can access symbols and functions which are explicitly exported by the kernel code through specific macros.

increased considering the dscp marking color. Subsequently, the outer IPv6+SRH is removed and the decapsulated packet is forwarded to the recipient. On the contrary, if the flow is not found then the packet is passed to the Linux kernel networking stack for being processed as usual.

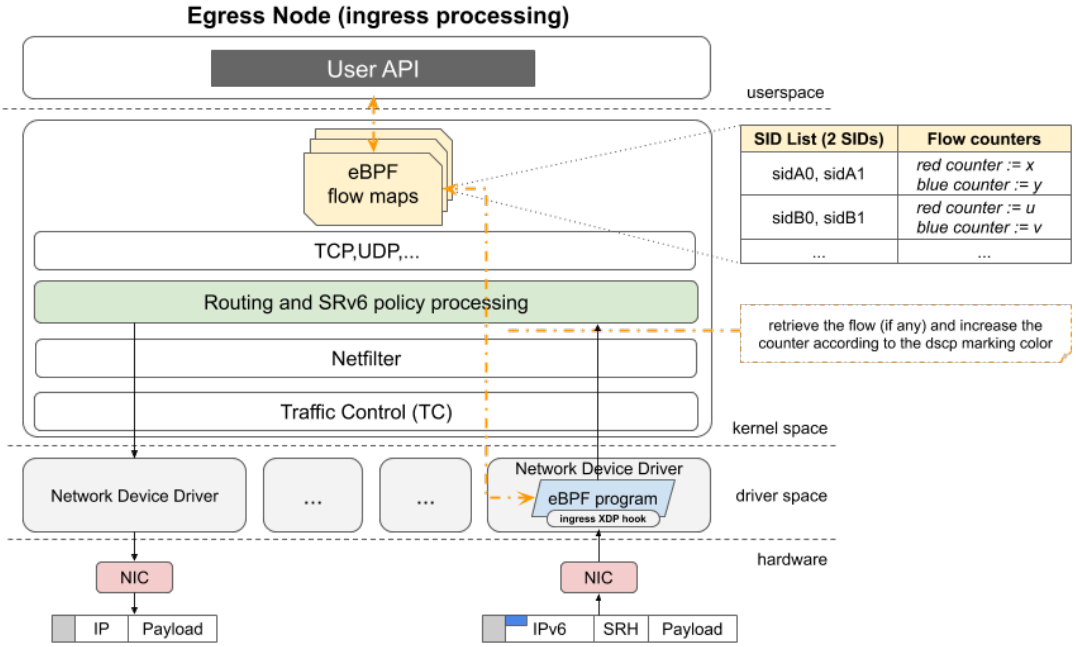


Figure 5.7: *Packet processing and statistics management in the Egress node using the PF-PLM eBPF implementation.*

In Figure 5.8, I illustrate the processing and management of statistics in the Ingress node realized with the eBPF program named `tc_srv6_pfplm_egress` and attached to the TX hook offered by the Linux Traffic Control (TC). When a packet is received, the destination address (DA) is taken into account to find out if there exists a SRv6 policy to be applied. If a SRv6 policy is found, the SRv6 networking subsystem enforces such policy encapsulating the packet within an IPv6+SRH packet. Otherwise, the packet continues to be processed by the Linux networking stack which does not apply any IPv6+SRH encapsulation. However, before any packet is sent over the network, the `tc_srv6_pfplm_egress` eBPF program intercepts it by leveraging the eBPF egress TC hook[4]. This eBPF program looks for the SRH among the IPv6 header extensions. If the packet does not come with a SRH, no further actions are taken and it continues through the Linux networking stack. Instead, if the SRH is present then the *SID List* length is used to retrieve the suitable eBPF flow map. The *SID List* is matched against the entries of the flow map in order to determine whether that list corresponds to a flow to be monitored. If no matches are found, the eBPF program does not process the packet further and it returns the control to the Linux kernel networking stack. On the contrary, the flow counters are retrieved and increased considering the active color obtained from the eBPF color map. The dscp of the outer IPv6 packet is also changed reflecting the active color and the packet is transmitted over the network.

[4]There is no XDP hook that enables eBPF programs to process packets which are about to be transmitted over the network.

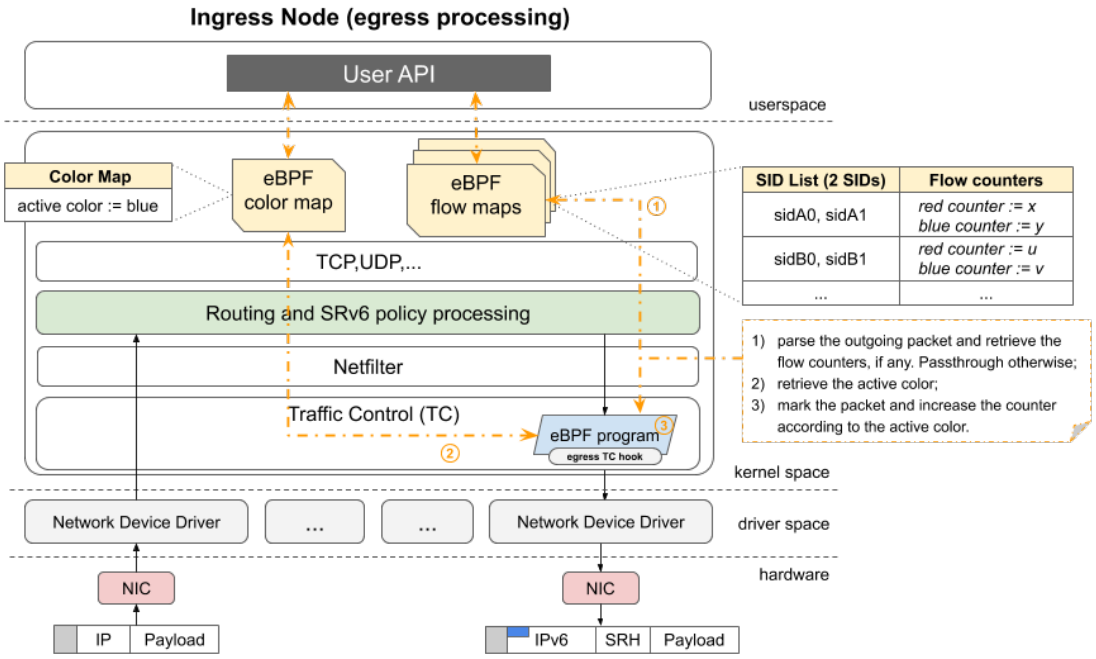


Figure 5.8: *Packet processing and statistics management in the Ingress node using the PF-PLM eBPF implementation.*

5.5.5 TWAMP Sender and Reflector

An SRv6 version of the *TWAMP Sender* and *Reflector* has been implemented using the Python language for testing the counting system of the different PF-PLM implementations. The two components realize the procedures for collecting counters and periodically changing the active color by integrating specific drivers to deal with the several PF-PLM implementations.

The TWAMP messages have been implemented using the Scapy project [143], a python library for sending and receiving packets including SRv6. The measurement procedure is initiated by the *Sender*, which reads the local counter and generates the query packet that is sent using the SRv6 path. The *Reflector* receives the packet, reads the missing counters and sends the response packet back to the Sender, which is eventually able to evaluate the packet loss. Both *TWAMP Sender* and *Receiver* are interfaced to the SDN controller through the Southbound gRPC which which implements two services: i) management of SRv6 entities like path and behaviors; ii) operations to start/stop the performance measurements and to retrieve/collect counters. Further details about the Southbound gRPC API are available in [135], while the open-source implementation of the python code can be accessed online [158].

5.6 SRv6-PM testbeds and experiments

Several tools have been developed for validating the SRv6-PM platform as well as the different implementations of the accurate packet loss monitoring system. SRv6-PM is made up of two parts: i) the SDN controller and management plane along with the cloud-native architecture for Big data processing; ii) the SRv6-PM data plane with the several PF-PLM implementations. In Subsection 5.6.1, I briefly introduce the tools

used to help developers easily deploy the SDN controller and cloud-native components for processing collected measurements. In Subsection 5.6.2, I describe the testbed I created to evaluate the performance of the PF-PLM implementations discussed so far.

5.6.1 Reproducing the experiments

There are different approaches for deploying the cloud-native workload implementing the SDN control and management planes of the SRv6-PM platform: i) a "manual" and "static" "dockerized" solution; ii) a Kubernetes-based "automatic" and "dynamic" solution. The Docker-based approach is enough for the plain replication of the whole platform where multiple components are aggregated in a deployment unit to be run using the Docker Compose. Such an approach is preferred to become familiar with the entire SRv6-PM platform. Anyway, this is not the best way to deploy a cloud-native workload since it is not very flexible and it might require considerable effort in case of reconfiguration and extensions. To this regard, the Kubernetes-based [81] approach kicks. In particular, *Kubernetes Playground* [82] has been extended for deploying the cloud-native part of the SRv6-PM with the aim of simplifying both provisioning and configuration of virtual environments as well as workload orchestrations. Details about the SDN controller deployment and management plane are available in [135].

5.6.2 Cloudlab testbed for processing load evaluation

On Cloudlab facilities [33], I set up a testbed on bare-metal servers for evaluating the overhead introduced by the PF-PLM solutions in the Linux kernel networking stack. This testbed is the same as described in Chapter 3, Section 3.4 and it is made of two nodes which are connected back-to-back in a loop fashion. The Traffic Generator and Receiver (TGR) node sends traffic to the System Under Test (SUT) node where the PF-PLM solutions are deployed. Hence, the SUT processes the received packets and sends them back to the TGR. The SUT node runs a vanilla version of the Linux kernel 5.4 that I configured in two different ways: i) the SUT is configured as the Ingress node of the SRv6 networks, i.e. it carries out encap operations; ii) the SUT is configured as the Egress node, i.e.: it carries out decap operations.

I leveraged the SRPerf framework (see Chapter 3) to evaluate the maximum throughput that can be processed by the SUT when the PM solutions are deployed and turned on/off. In all my tests, the maximum throughput is defined as the maximum packet rate at which the packet drop ratio is smaller than or equal to 0.5%. This is also referred to as Partial Drop Rate at a 0.5% drop ratio (in short PDR@0.5%). For further details on the methodology, please refer to Chapter 3, Subsection 3.3.2.

5.7 Performance results for PF-PLM solutions

To carry out the performance experiments on the SUT, I crafted IPv6 UDP packets encapsulated in outer IPv6 packets (78 byte including all headers). The outer packets have an SRH with a *SID List* of one SID. I considered two separate classes of tests where different PF-PLM implementations have been deployed: i) *Iptables* vs *IP set*; ii) *IP set* vs *eBPF*. I repeated each test four times; as described in Chapter 3, each test includes a large number of experiments and repetitions to estimate the maximum throughput.

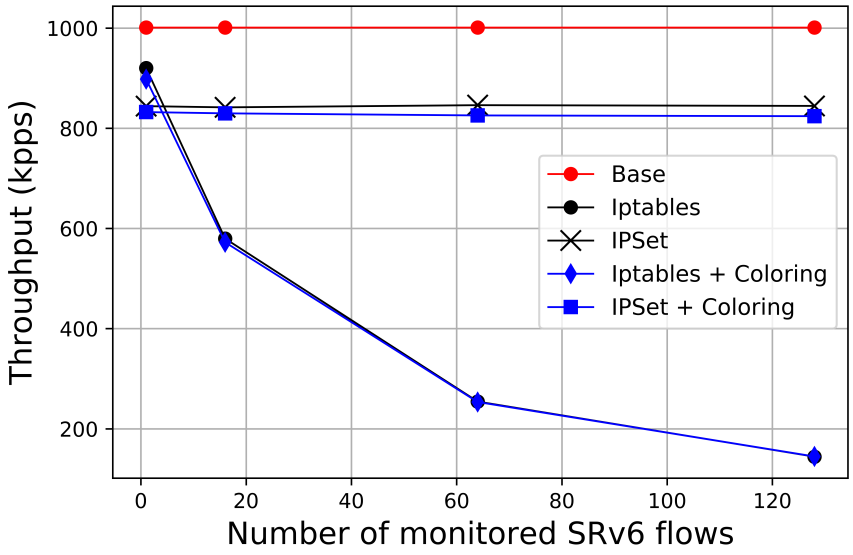


Figure 5.9: *SUT throughput (Ingress node configuration: Iptabels vs IP set performance comparison).*

5.7.1 Processing load of packet counting: *Iptabels* vs *IP set* solutions

In Figure 5.9, I plot the throughput (in kpps) for the SUT configured as an Ingress node considering different settings. In this case, I compared the *Iptables* against the *IP set* implementations. The red line represents the SUT base performance in the simple SRv6 case, i.e. it applies SR policies but does not perform any counting or coloring operations. In this case, the measured average maximum throughput reached is about 995 kpps. Then, I considered the performance loss due to the counting operations for both the *Iptables* and the *IP set* solutions, varying the number of monitored flows. The SUT performs the matching operation but does not modify the packet to color it. The experiments results are reported in Figure 5.9 using the black lines. As expected, using the *Iptables* based PF-PLM the performance degrades increasing the number of monitored *SID Lists* (i.e. the number of *Iptables* rules). The measured throughput decreased in an inversely proportional way with the number of required *Iptables* rules. In contrast, the *IP set* based solution result in a throughput that is almost constant with the number of monitored *SID Lists*. By observing Figure 5.9, the *Iptables* based implementation achieves a higher SUT throughput with respect to the *IP set* based one when a single flow needs to be monitored. In particular, the throughput degradation compared to the base case is 8%, while the degradation for the *IP set* based PF-PLM is 15.5%. However when 16 flows are monitored the throughput of the *Iptables* based PF-PLM decreases by 42%.

Finally, I evaluate the coloring cost of both solutions. In this case, the degradation both in the *Iptables* and in the *IP set* implementations is less than 2.5%. In particular, in the *Iptables* case the maximum degradation is measured when a single rule is present and it reaches 10.3% with respect to the base case. The coloring loss in the *IP set* case is slightly less and the throughput degradation with respect to the base case is about 17.5%.

In Figure 5.10, I report the same analysis for the SUT configured as an Egress node.

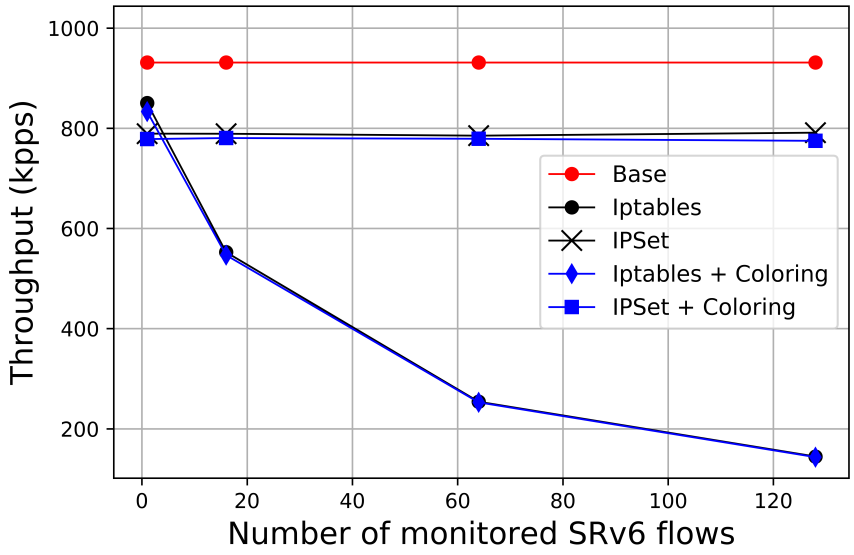


Figure 5.10: *SUT throughput (Egress node configuration: Iptabels vs IP set performance comparison).*

It can be noted that the decapsulation operation is more demanding and the total overall throughput in the base case is lower (about 940kpps). As for all the other configurations, the trend is similar to that of the ingress node with similar percentage degradation.

5.7.2 Processing load of packet counting: *IP set* vs *eBPF* solutions

On the Ingress node, the eBPF program hooked to TC provides better performance in terms of the overall throughput when compared to the performance achieved with the

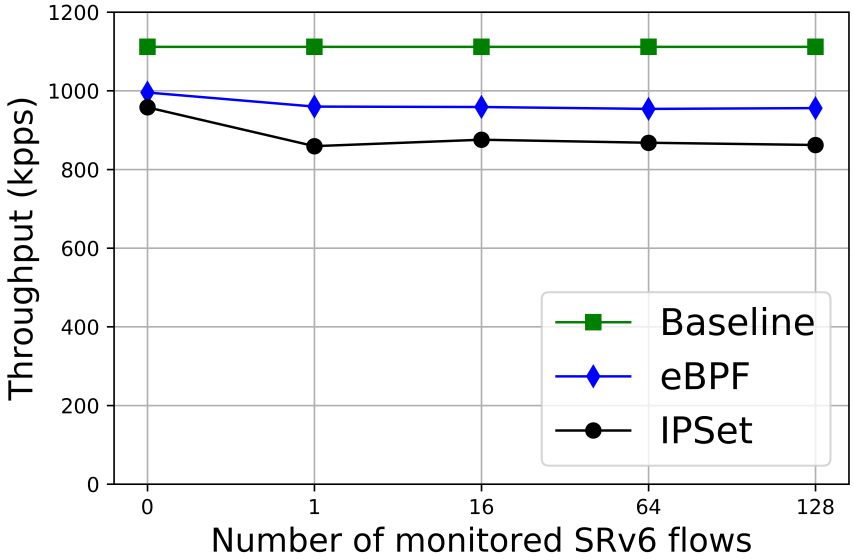


Figure 5.11: *SUT throughput (Ingress node configuration: IP set vs eBPF/TC performance comparison).*

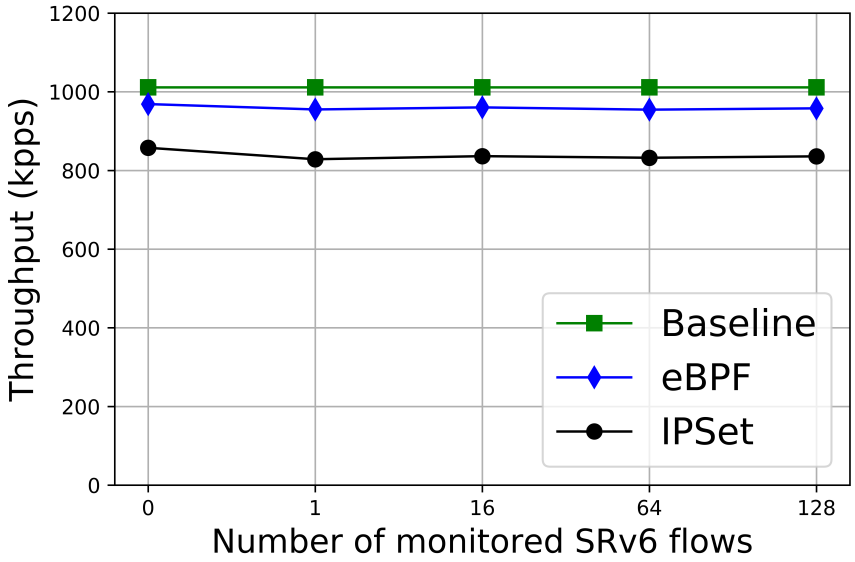


Figure 5.12: *SUT throughput (Egress node configuration: IP set vs eBPF/XDP performance comparison).)*

IP set based solution. As shown in Figure 5.11, the *eBPF* based PF-PLM starts with a 10.5% degradation (w.r.t. the baseline) when there are no flows to be monitored. The overall performance degradation reaches the average value of 13.9% for one flow to be monitored and this value remains stable regardless of how many additional flows are added. The *IP set* based PF-PLM starts with a 13.8% performance degradation for zero flows to be monitored and it reaches the stable value of 22.0% for one or more monitored flows. This is due to the low processing overhead of the TC layer compared to the non-negligible one introduced by the Netfilter framework on which many running subsystems are hooked, i.e. *Iptables* and all the related tables and chains.

On the Egress node, the performance achieved with the eBPF program is considerably better if compared to the performance of the *IP Set* module. As reported in Figure 5.12, the *eBPF* based PF-PLM achieves a 4.2% performance degradation (w.r.t. the baseline) in case there are no flows to be monitored. The performance drops at 5.4% on average if there is one flow to be monitored and it keeps stable no matter how many flows (to be monitored) are subsequently added. The *IP set* based PF-PLM introduces a 15.0% of degradation for zero flows and it remains stable at 17.6% for one or more flows to be monitored. The reason lies in the fact that the eBPF program is directly hooked to the eXpress Data Path (XDP), and in this case the network card driver supports the XDP native-mode. Native mode executes eBPF/XDP program in the networking driver early RX path. At this stage, a network packet is nothing more than a mere sequence of bytes with no associated metadata. Raw data access is very efficient, as the operations needed for parsing the packet headers and for extracting the flows are not so computationally expensive.

5.8 Conclusions

In this Chapter, I proposed an architecture that I contributed to design and develop for supporting Performance Monitoring of SRv6 networks. The architecture relies on open source components which have been used for building the cloud native part for management, the control plane as well as the specific data plane part focused on Loss Measurements in SRv6 networks. With regard to the data plane, I focused on an accurate Per-Flow Packet Loss Measurement (PF-PLM) framework based on the extension of the TWAMP protocol and the alternate marking techniques. I have designed and realized several PF-PLM solutions which rely on the Linux kernel networking stack. In particular, I have implemented the marking and packet loss counters leveraging different packet processing frameworks inside the kernel. My first solution consisted in an extension of the Netfilter/Xtables/Iptables framework. By setting up an evaluation platform, I was able to determine the cost of activating the monitoring, i.e. the degradation of the maximum forwarding throughput achieved in a node. As already noted during the design phase, the overhead of the *Iptables* based implementation increases as the number of SRv6 flows to be monitored grows. For this reason, I designed and implemented another solution that leverages the *IP set* framework still based, under the hood, on Netfilter. However, this time the overhead introduced by this monitoring solution does not depend on the number of the monitored SRv6 flows. Even though the *IP set* implementation scales as the number of monitored flows increases, the overall overhead introduced by the monitoring is roughly about 15.0% which is by no means negligible. At this point, I completely changed my perspective and moved to a different design based on a new technology that exploits an in-kernel virtual machine (eBPF VM) able to run network programs. With this approach, I was able to achieve a significant performance increase over the *IP set* implementation, getting closer to the basic maximum throughput achieved for a flow that is not monitored. Indeed, the overall overhead introduced by the PF-PLM *eBPF* based solution is less than 5.0%.

I believe that the SRv6-PM open source platform represents a valuable and re-usable tool for further development and improvement of Performance Monitoring in SRv6 networks. This platform can be used to assist in the process of standard definition, by allowing the early prototyping and testing. For example, I am planning the development of SRv6 Delay Monitoring, which has also been proposed in the standardization.

CHAPTER 6

An HybrId Kernel/eBPF data plane framework for boosting up performance in SRv6 based Hybrid SDN networks

The work I present in this Chapter is mostly taken from my paper *Performance Monitoring with H^2 :HybrId Kernel/eBPF data plane for SRv6 based Hybrid SDN*, submitted to *Elsevier Computer Networks Special Issue on Challenges and Solutions for hybrid SDN, 2021* [2].

6.1 Introduction

Segment Routing (SR) [54, 148] brings Software Defined Networking (SDN) [38] into the networks of service providers, going beyond its application into Data Center Networks. According to a vendor's statement, SR is the "de-facto SDN Architecture" [149]. The SR architecture has been implemented with MPLS data plane (SR-MPLS) [4] and with the recent IPv6 data plane (SRv6) [25]. In this Chapter, I will foster the SRv6 implementation which is also attracting a lot of interest from the industry, academia and whereby there are several on-field deployments [147].

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. As I have already thoroughly explained in Chapter 1, 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 this context, the *network program* corresponds to an *SR Policy* which consists in a list of *Segments* (SIDs), i.e. a list of IPv6 addresses. In the majority of 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 (*SID 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 [122] based SDN where the SDN controller needs to install flows also in the core nodes.

It is worth noting that the so-called SD-WAN solutions [145] share the same philosophy as HSDN/SRv6. In the SD-WAN approach, 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 SRv6 also makes it 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) [40].

In this Chapter, I consider a Linux-based software router in an HSDN/SRv6 architecture exploiting the extended Berkeley Packet Filter (eBPF) [108], the eXpress Data Path (XDP) [70] and the Traffic Control subsystems. Hence, I propose a data plane architecture called HIKe-v0, which stands for HybrId Kernel/eBPF forwarding (version 0)[1]. The HIKe-v0 architecture highlights the need for integrating the packet forwarding and processing based on a "traditional"[2] 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, I envisage an additional "hybrid" approach "inside" the forwarding engine, i.e. based on a flexible and modular combination of the Linux kernel, eBPF/XDP bypass and eBPF/Traffic Classification (TC). Therefore, by considering the two different conceptual levels of hybridization, I will call the overall solution *Hybrid squared* or H^2.

The aim of HSDN networks is 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. At the same time, 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 provide complex services such as SFC, Tunneling/VPN, Monitoring, etc. Furthermore, I aim to devise an architecture that takes advantage of the eBPF efficiency 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, I focus on Performance Monitoring (PM) in Hybrid SDNs. To this regard, I consider the need to implement accurate and real-time per-flow monitoring of QoS parameters (delay, jitter, loss). As I demonstrated in Chapter 5, implementing an accurate Per-Flow Monitoring solution severely impacts the performance of the data plane and,

[1]Currently, I have designed and implemented two versions of the HIKe architecture which are respectively the HIKe-v0 and HIKe-v1. I prefer to treat the two versions separately since the v1 introduces significant improvements over v0.

[2]By the term "traditional" I refer to the Linux kernel networking stack that does not take advantage of any eBPF virtual machine-based packet processing capability.

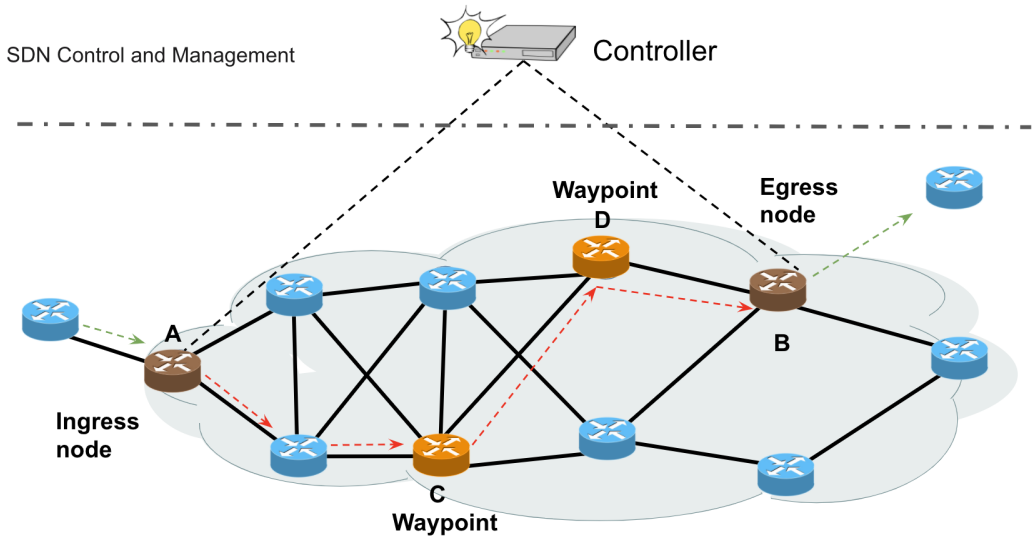


Figure 6.1: *Reference HSDN Network scenario over IPv6 based networks.*

therefore, requires the most efficient implementation possible. I demonstrate how the HIKe-v0 data plane architecture supports the Per-Flow Packet Loss monitoring with a very limited impact on performance. This data plane solution can be deployed in the SRv6-PM architecture (see Chapter 5) with the aim of significantly increasing the performance of the whole monitoring system.

The following Chapter is organized to emphasize my contributions as follows. In Section 6.2, I first introduce the hybrid SDN model based on the SRv6 technology (HSDN/SRv6) from a system architecture viewpoint using the network Performance Monitoring as a reference case. Then in Section 6.3, I describe the Linux data plane for HSDN/SRv6, focusing on the Kernel based forwarding, the eBPF/XDP and eBPF/TC packet processing subsystems. In Section 6.4, I devise my first version (v0) of the hybrid data plane architecture called HIKe-v0. I implemented a first open source proof-of-concept of HIKe-v0 and I measured the upper bounds of its performance as described in Section 6.5. In Section 6.6, I analyze the related work. Finally, I draw some conclusions in Section 6.7.

6.2 SRv6 as HSDN Solution

In this Section, I am going to briefly discuss how the SRv6 technology is used in an HSDN architecture. For a comprehensive discussion about Segment Routing over IPv6 networks, please refer to Chapter 1.

6.2.1 HSDN Network Scenario

For the sake of clarity, I am going to consider the reference network depicted in Figure 6.1. Node A and Node B represent respectively the ingress and the egress node to an SRv6 domain. At the edge of the network packets are encapsulated in SRv6 traffic flows, i.e. the ingress node adds an outer IPv6 header to the packets that can include a Segment Routing Header (SRH) carrying the *SR Policy*. In this example, the first two SIDs (IPv6 addresses) included in the *SID 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 at the egress edge node, the inner IP packet is decapsulated, i.e. the outer IPv6 header and SRH are removed.

6.2.2 SRv6 Networking programming and HSDN

The SRv6 network programming model (see Chapter 1) defines packet processing in network nodes as the execution of a (network) program whose instructions are represented by the SIDs carried in the SRH. Such instructions can range from simple operations of forwarding to complex ones such as encap/decap or packet counting. Network programming introduces an additional level of flexibility for the HSDN network. In fact, in SRv6 networks it is not always necessary to control all nodes as it is often enough to control the ingress/encapsulating node (i.e. the one which applies the *SR Policy*) to activate functions along the path of a packet. Therefore, if it is required to measure performances of a flow or even of a single packet, the controller can interact with the ingress node by inserting in the *SID List* all the operations to be carried out at the different nodes along the network path.

The network programming approach provides 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.

6.2.3 SRv6 Southbound Interface

At the time of writing, a standard Southbound Interface (SBI) for SRv6 HSDN networks has not been defined yet. Few proposals for SBI are available in the literature and the most comprehensive one is presented in [172]. Here, the authors discuss the actual needs of SRv6 nodes and also compare them with SDN enabled routers that support the traditional Openflow interface.

Considering the Performance Monitoring context, the Hybrid Kernel/eBPF data plane solution discussed in this Chapter can be controlled with the same SDN controller presented in Chapter 5, since I designed the HIKe-v0 solution to be fully compatible with the HSDN/SRv6 architecture deployed in SRv6-PM.

6.2.4 Performance Monitoring in HSDN/SRv6 networks

Performance Monitoring (PM) standardization activities for SRv6 are still ongoing and the TWAMP light protocol [58] seems to have reached a promising level of maturity. TWAMP technique can be effective for collecting some metrics such as packet loss, one-way and two-way packet delays following the “fate” sharing paradigm. Data collection is accomplished via special test UDP packets transmitted over the measured path, which are then handled by the node requesting the measurement. TWAMP based PM supports the Alternated-marking [56], [113] method for accurate loss monitoring which can be applied to IPv6 and SRv6 flows. For further details about the PM standardization and solutions, please refer to Chapter 5, Section 5.2.

6.3 Linux SRv6 Networking, the eBPF/XDP fast path and the eBPF/TC

In this section, I first introduce the envisage application scenarios for Linux HSDN/SRv6 enabled routers. Next, I briefly present some details of SRv6 networking implementation approaches focusing on eBPF/XDP and eBPF/TC.

6.3.1 Linux HSDN/SRv6 Applications

Considering the reference scenario described in the Section 6.2.1, there are several use cases where Linux can be used to implement several of the features required by HSDN/SRv6 enabled routers, especially at the network edge. The most relevant 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. Relying on this architecture, 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, a Linux based server 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 [176] (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.

Likewise, with NFV (Network Function Virtualization, see Chapter 4) it is very common to have Linux based virtual appliances.

Another 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).

6.3.2 Implementation aspects of Linux SRv6 Networking

The support for HSDN/SRv6 capabilities in a Linux based router can be achieved in different ways: i) SRv6 subsystem in the Linux kernel networking stack based on *LWT*s; ii) user space SRv6 implementation based on VPP capable of exploiting DPDK hardware acceleration; iii) eBPF Virtual Machine whose programs can be hooked in different *hook-points* within the kernel networking stack, i.e. XDP and TC hooks.

VPP is a kernel bypass solution which is intended to process packets in user space. By giving full control of the NIC to a user space program, the kernel overhead is reduced and it is relevant enough when working at speeds of 10Gbps or higher. However, user space 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, the first ("traditional" Linux kernel networking stack) and the third approaches (eBPF based) are the most appealing since I can propose a hybrid approach among these two. I need to point out that the eBPF based solution is actually integrated in the

Linux kernel, but for the sake of clarity I will use the term "traditional" (or "normal") kernel for referring to the approach that does not rely on any eBPF packet processing acceleration. eBPF/XDP is extremely good for performance reasons, but is it possible to 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, however, 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 is dynamically updated by the control plane inside an eBPF program.

6.3.3 Packet processing through the eBPF Virtual Machine (VM) and Linux kernel hooks

In Chapter 2, I have already introduced the extended Berkeley Packet Filter (eBPF) and how it is leveraged in the Linux kernel for packet processing purposes. For the sake of clarity, I only summarize the main aspects about eBPF hereafter, focusing my attention only on the packet processing aspects. The extended Berkeley Packet Filter (eBPF) is a revolutionary technology with origins in the Linux kernel that can run sandboxed programs (through the eBPF Virtual Machine) in a privileged context such as the operating system kernel. It is used to safely and efficiently extend the capabilities of the kernel without requiring to change kernel source code or load kernel modules. By allowing to run sandboxed programs within the operating system in an event-driven fashion, application developers can run eBPF programs to add additional capabilities to the operating system at runtime. Those eBPF programs are guaranteed to be safe from the kernel once they have been verified and loaded by the kernel. To satisfy safety requirements, eBPF programs undergo several constraints which are, for example, related to the maximum of instructions to be supported, avoidance of unbounded loops, no access to kernel internals unless through the use of a restricted number of specific helper functions and so on.

Regarding the packet processing, eBPF programs can be attached in several predefined hooks placed in the Linux kernel. The most interesting hooks are the ones offered by the eXpress Data Path (XDP) and the Linux Traffic Control (TC). XDP, proposed in [70], is an eBPF based high performance packet processing component merged in the Linux kernel since version 4.8. XDP introduces an early hook in the RX path of the kernel (at the NIC driver space) and before any memory allocation takes place, ensuring the highest performance possible. Unfortunately, eBPF programs attached on the XDP hook can only process incoming packets and not the ones that are leaving the node. For these reasons, other eBPF hooks are available in the kernel: the Traffic Control offers two more hooks which are respectively the TC ingress hook and the TC egress one. Since TC hooks are only available after a packet has been fully handled by the kernel (the `sk_buff` has already been allocated), the performance experienced by TC hooks is not as good compared to XDP performance. However, the TC egress provides the last hook that an eBPF program can be attached to for processing a packet that is about to be transmitted over the network.

6.4 The HIKe-v0 data plane: Hybrid Kernel/eBPF (version 0)

The aim of HIKe-v0 data plane is to combine the conventional Linux kernel networking stack with eBPF programs hooked into XDP and TC, in a flexible and modular way and taking the best from both worlds. An eBPF program allows to process packets introducing a very small overhead, 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 "traditional" Linux kernel networking stack is slower but it offers full control on the way in which packets have to be processed.

6.4.1 Overview

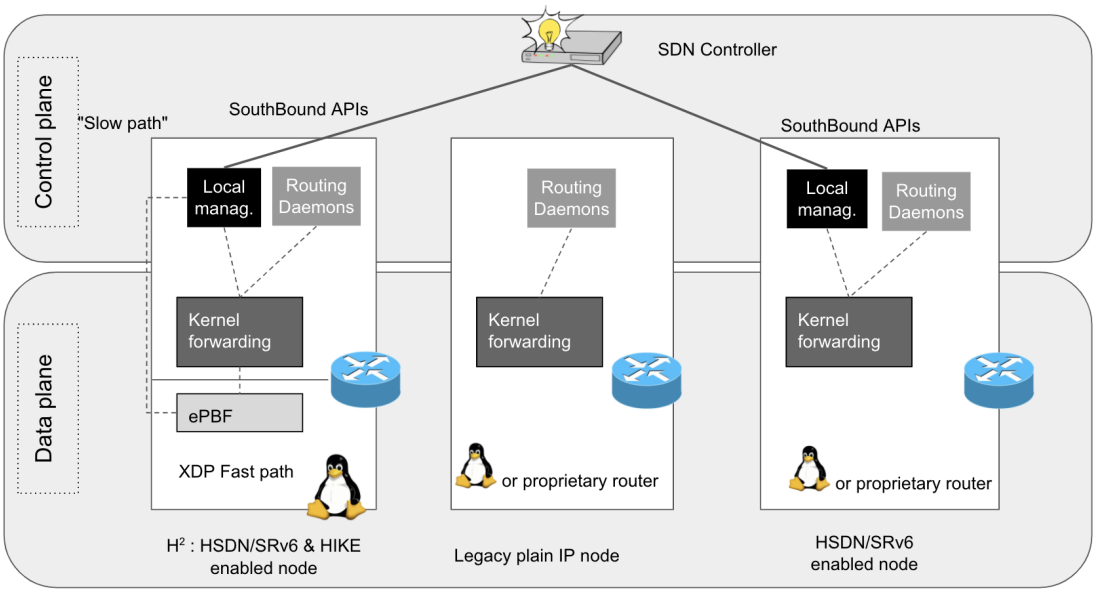


Figure 6.2: *High level overview of the HIKe-v0 data plane/control plane architecture.*

In Figure 6.2, I outlined the architectural overview of the different components of the proposed architecture. Each node is a Linux based device where the packet processing functionality is logically divided between a *kernel path* and *fast path*. The fast path includes eBPF programs that take advantage of the XDP hook (within the NIC driver) in order to provide efficient and high-performance packet processing. The kernel path relies on the traditional routing and forwarding capabilities implemented by the Linux kernel networking stack whose routing tables can be configured by hand or by using any routing daemon. The Linux Traffic Control (TC) is a subsystem of the kernel path which aims to offer shaping and control features, i.e. it gives the ability to configure the kernel packet scheduler. In addition, TC offers two hooks (for incoming and outgoing packets) that are used to hook eBPF programs in order to process network traffic.

Southbound APIs connect a Local Manager module with the SDN controller. The Local Manager is responsible for implementing the custom logic, configuring programs and behaviors 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 6.2 also highlights the difference among 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.

6.4.2 Full programmable node

In Linux networking, conventional systems such as Netfilter/Xtables/Iptables offer a relatively high degree of modularity, allowing different filtering and processing tasks 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[3]), the set of the kernel functions that can call inside an eBPF program (known as eBPF helper functions) are severely limited, and most importantly, eBPF is a non-Turing complete language (i.e. 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 (eBPF) programs implementing basic functionalities and then combine them flexibly. However, the number of eBPF programs that can be concatenated is limited (i.e. it is possible to chain a max of 32 programs) and in addition this solution requires manual handling of program and high technical skills. For these reasons, eBPF programs have been often used mainly as fast but standalone/monolithic packet processors.

HIKe-v0 proposes a way to enhance eBPF/XDP and eBPF/TC, to support modularity and foster efficient program 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 Subsections 6.4.3 and 6.4.4) 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* (see Chapter 2, Section 2.3.

In the discussed Performance Monitoring scenario, the first eBPF program to be called is determined by considering (matching) the IPv6 destination address of a packet. However, in general this is applicable to an arbitrary set of matching rules and packet fields.

6.4.3 HIKe-v0 eBPF/XDP architecture

Figure 6.3 shows the eBPF/XDP architecture for HIKe-v0. As can be observed, packets enter from the RX port and are passed to the XDP hook. Once processed, packets exit from HIKe-v0 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, I draw a dividing line in Figure 6.3: the components below the line stand in the kernel NIC driver space and benefit of the fast eBFP/XDP processing.

[3]The limit was increased up to 1M instructions for privileged users.

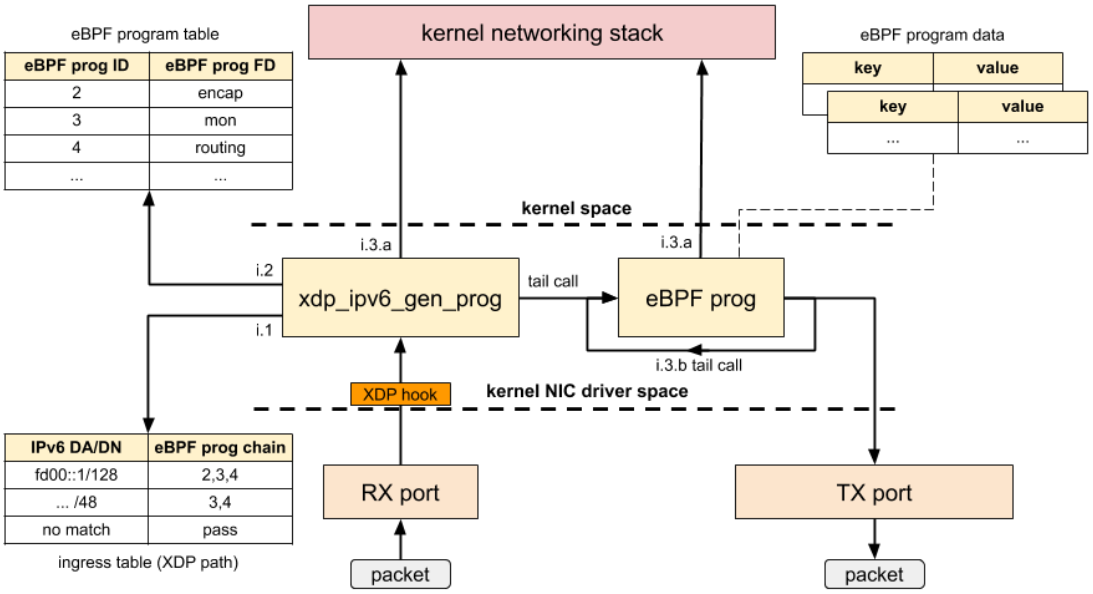


Figure 6.3: *HIKe-v0 eBPF/XDP architecture.*

Packets are handled by the `xdp_ipv6_gen_prog` eBPF program [4] that performs an initial lookup (i.1) in the `ingress table` and, according to some packet features (i.e. the IPv6 destination address network in this 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 according 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 values. If there is no associated eBPF program chain to a specific IPv6 destination address, the framework can provide a default eBPF program chain (if defined) or the packet is passed directly to the "traditional" Linux kernel networking stack.

When an eBPF program has to be executed, the HIKe-v0 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*. The `xdp_ipv6_gen_prog` could be seen as a "trampoline" which parses the IPv6 packet, retrieves the eBPF program chain and transfers the control flow to the first eBPF program in the chain.

An HIKe-v0 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-v0 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*

[4] 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.

calls allowed in eBPF, which is currently set to 32.

HIKe-v0 eBPF programs are stateless but they can store and retrieve information in dedicated eBPF maps (as depicted in Figure 6.3). They could also rely on some already defined context variables which can be conveniently used to speed up implementation and processing. These variables (i.e. the offset at which the IPv6 header starts) allow clock cycles to be saved, for instance by avoiding parsing the packet twice. Passing such a 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, I 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-v0 framework, thanks to the use of chains it is possible to reuse and combine eBPF programs depending on the type of traffic that needs to be processed. HIKe-v0 provides a number of prefabricated eBPF programs such as routing, SRv6 encap/decap and monitoring.

The routing in HIKe-v0 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 kernel networking stack. The second routing program is based on a hybrid eBPF/XDP + kernel approach since 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.

To contextualize the HIKe-v0 operations carried out in the XDP context, it is worthwhile to exemplify by considering the ingress node of an SRv6 network (Node A in Figure 5.2). This node 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 where 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 2,3,4 (`encap`, `mon`, `routing`). The program `xdp_ipv6_gen_prog`, leveraging a HIKe-v0 helper function, starts the *tail call* sequence by running the `encap` program that encapsulates the IPv6 packet into another IPv6 packet with the Segment Routing Header (SRH). After the encapsulation (`encap`), the program that performs the monitoring (`mon`) tasks is invoked (i.e. counting and marking the packet) and finally the program that operates the routing (`routing`) determines the next hop and the associated output interface.

6.4.4 HIKe-v0 eBPF/TC egress hook

When the HIKe-v0 fast path cannot process a packet, the entire processing is on the behalf of the Linux kernel 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

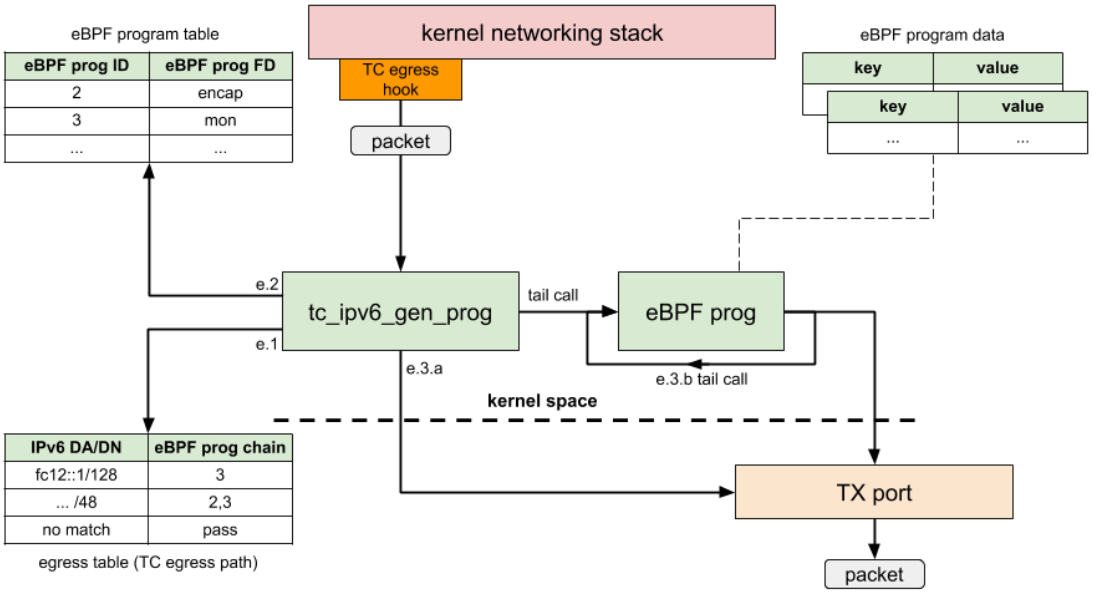


Figure 6.4: *HIKe-v0 eBPF/TC architecture.*

and stability. Thus, the Linux kernel introduces the TC egress hook where eBPF programs can be attached. Aside from 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 possible to perform some additional processing on the packets that are generated locally or forwarded to another hop. Therefore, in the HIKe-v0 architecture, I introduced another programmable framework for packet processing, exploiting the TC egress hook to attach an eBPF program. The HIKe-v0 TC architecture reported in Figure 6.3 is very similar to the eBPF/XDP one 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 chain 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-v0 eBPF/XDP framework cannot be leveraged in the HIKe-v0 eBPF/TC one and vice versa. This implies that, for instance, the `encap` eBPF/XDP program and the `encap` eBPF/TC one are two distinct programs but they share some internal libraries and helper functions made available by the HIKe-v0 framework for avoiding code duplication as much as possible.

6.4.5 Monitoring System Implementation

To assess the performance of the HIKe-v0 architecture in a real scenario, I provide another implementation of the Per-Flow Packet Loss Monitoring (PF-PLM) solution presented in Chapter 5. The PF-PLM implementations that I provided in Chapter 5 are based on Linux kernel modules (i.e. *Iptables* and *IP set*) and on ad-hoc eBPF programs hooked into XDP and TC. While such programs are built exclusively for the SRv6-PM architecture, the eBPF programs shipped with the HIKe-v0 can be reused for any other

purpose which is not strictly related to the performance monitoring. To carry out all the necessary operations required for the packet loss monitoring use case (i.e.: parsing SR policies, updating flow counters, coloring), I designed and implemented the HIKe-v0 eBPF programs so that they can be used in a stand-alone fashion and/or chained together with other HIKe-v0 eBPF programs through program chains.

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

The HIKe-v0 implementation I used for performance evaluation is available at [67]. An implementation of the full performance monitoring demonstrator is available as part of the ROSE ecosystem [137].

6.4.6 HIKe-v0 limitations

The HIKe-v0 architecture moves the first step in the direction of creating a framework for simplifying and reusing as much as possible the eBPF programs developed so far. In the HIKe-v0 architecture, an eBPF program chain is a very simple *object*: it contains the list of the HIKe-v0 eBPF programs that can be invoked one after the other to carry out a given processing logic on a packet. In other terms, once the program chain has been identified by the "trampoline" program, the processing logic starts from the first eBPF program in the chain up to the last one. In other words, an eBPF program chain can be seen as a "program" made of only "call" instructions which are encoded as the ID of the eBPF programs that are going to be called in turn to process the packet.

Since a program chain supports only a sequence of "call" instructions, it becomes impossible jumping to specific programs contained in the chain depending on the execution context, i.e. the packet comes with a specific header and, therefore, a given eBPF program must be executed in the chain rather than another. This approach leads to an important limitation in the HIKe-v0 processing model: the processing logic must necessarily follow the order in which the eBPF programs are specified in the chain (as a list) without the possibility of altering, in any way, their execution order. To overcome this limitation, I have designed and implemented a new version of HIKe, named HIKe-v1, that allows to encode different types of instructions inside a chain. Such instructions can be arithmetic/logic, conditional and unconditional jump, program call, etc. In this way, a chain becomes a "fully-fledged" program where individual eBPF programs represent composable processing functions to be invoked depending on the business logic of that chain and/or on the execution context.

6.5 HIKe-v0 performance results

In this Section, I present and discuss the performance of the HIKe-v0 implementations. The purpose of the following experiments is to assess the impact of HIKe-v0 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 testbed aims to evaluate the impact of these operations which are built on top con conventional router operations.

6.5.1 Processing load evaluation

I set up a testbed on Cloudlab facilities [33] for evaluating the processing load introduced by the HIKe-v0 architecture. The testbed is an exact copy of the one that I already described in Chapter 3, Section 3.4 and it is made of two nodes which are connected back-to-back in a loop fashion. The Traffic Generator and Receiver (TGR) node sends traffic to the System Under Test (SUT) node where the HIKe-v0 framework and implementations (i.e. internal tests and Performance Monitoring) are deployed. Hence, the SUT processes the received packets and sends them back to the TGR. On the SUT node, I installed an Linux-based OS running a vanilla Linux kernel 5.6 that I configured to test different scenarios consisting of: i) the evaluation of the overhead introduced by HIKe-v0 during the *tail call* executions; ii) the evaluation of the Performance Monitoring system implemented thought the HIKe-v0 framework.

To evaluate the maximum throughput achieved by the HIKe-v0 solutions, I leveraged the SRPerf framework described in Chapter 3. In all my tests, the maximum throughput is defined as the maximum packet rate at which the packet drop ratio is smaller than or equal to 0.5%. This is also referred to as Partial Drop Rate at a 0.5% drop ratio (in short PDR@0.5%). For further details on the methodology, please refer to Chapter 3, Subsection 3.3.2.

6.5.2 HIKe-v0 *tail call* performance

On the SUT node, I first configured the HIKe-v0 architecture for analysing the performance as the number of the programs called with the *tail call* varies. Thus, I 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 estimate the maximum throughput that a software-based router can support, at the net of the other operations that must be added to handle packets (which necessarily imply a decrease in the maximum sustainable throughput by the node). In Figure 6.5, I plot the average maximum throughput vs the number of programs in the HIKe-v0 eBPF program chain. I executed 32 runs for each configuration, and I 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 I add a tail of 8 programs to handle the packet, the maximum average throughput drops to 9.7 mpps.

6.5.3 SRv6 Performance Monitoring

I assess the HIKe-v0 performance in terms of throughput for some SRv6 Performance Monitoring configurations. To this aim, I implemented optimized HIKe-v0 eBPF programs to execute i) the encapsulation operation (ENC); ii) the packet counting and marking operations (MON); and iii) the routing/forwarding task (FW). Therefore, I configured the SUT node to perform a combination of such programs, and specifically: i) only 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 calls*);

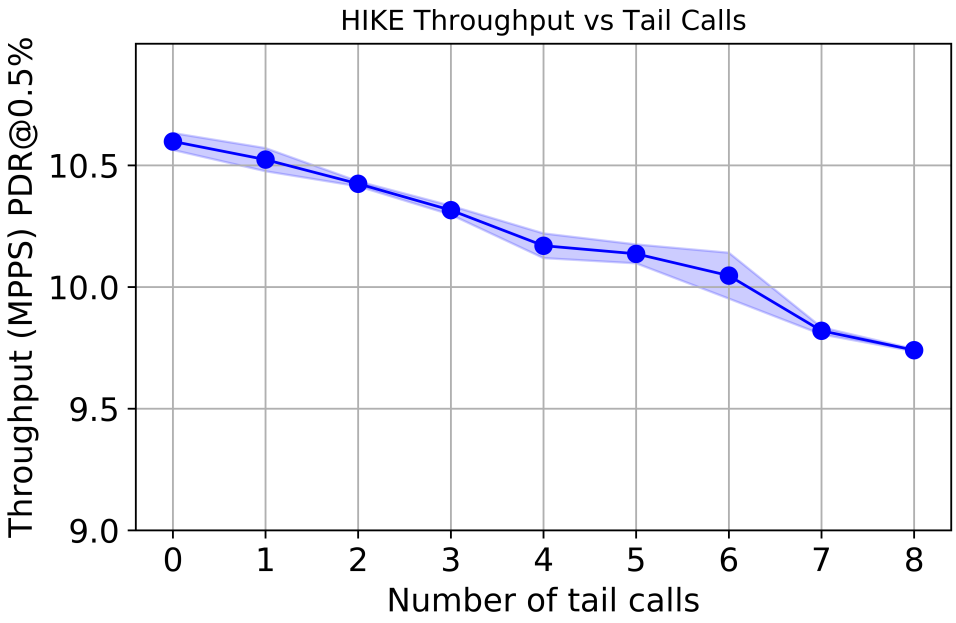


Figure 6.5: *HIKe-v0 eBPF program chain performance.*

iv) the encapsulation, monitoring and forwarding/routing tasks (3 *tail calls*). In Figure 6.6, I 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 networking stack. Moreover, I provided an upper bound implementing a *passthrough* 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 Chapter 5. 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 mpss, the configuration with encapsulation and forwarding/routing 3.1 mpss and the configuration with all the three operations combined 2.5 mpss. The difference between the different solutions lies in the number of accesses to the eBPF maps they perform.

By comparing the maximum throughput achieved with the forwarding/routing programs, is it possible to note a degradation from a baseline of $\sim$ 10 mpps attained by the basic HIKe-v0 architecture reported in Figure 6.5, 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 the current HIke-v0 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, I 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. I report the

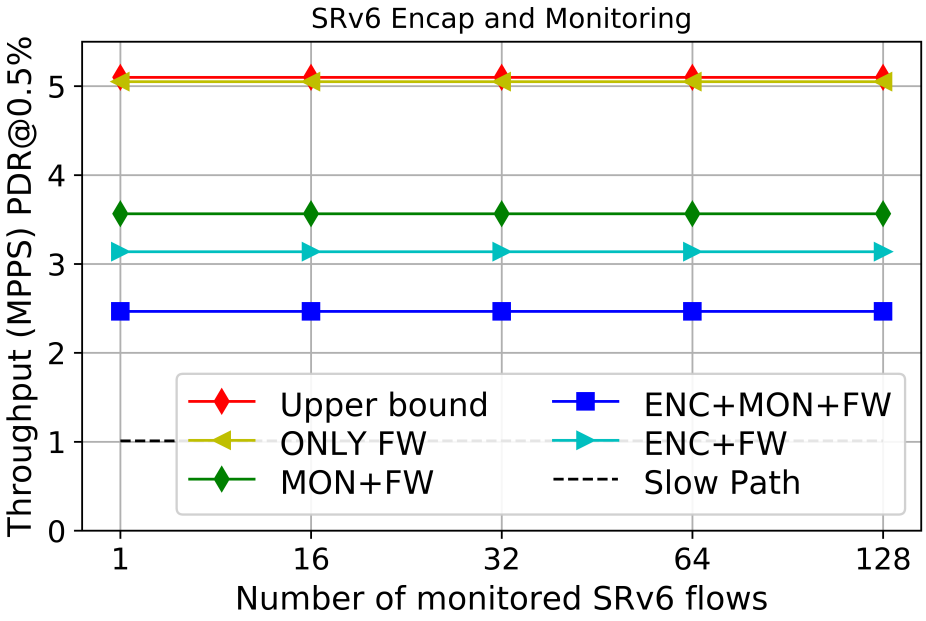


Figure 6.6: *HIKe-v0 fast path vs kernel forwarding performance.*

results in Figure 6.7. As can be observed, 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. The waypoint node can handle about 3.5 mpps with one SID and about 2.9 with 4 SIDs. Two trends emerge from this analysis: 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.

6.6 Related work

In this section, I first review Performance Monitoring solutions proposed for HSDN networks. Then, I present the related works on systems that leverage eBPF for HSDN networks.

6.6.1 Performance Monitoring in HSDN networks

Performance Monitoring of traffic flows in HSDN networks, relying on traditional techniques, poses several problems as highlighted in [6]. This is mostly due to the fact that it is difficult to integrate information from traditional nodes with the information collected in the softwarized network. For example, in [32] 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 engineering decisions. In order to reduce this problem, techniques such as *compressive sensing* can help to identify the main links of the network and focus monitoring operations on them to minimize estimation problems.

Solutions that are coming to prominence foresee the direct inclusion of monitoring

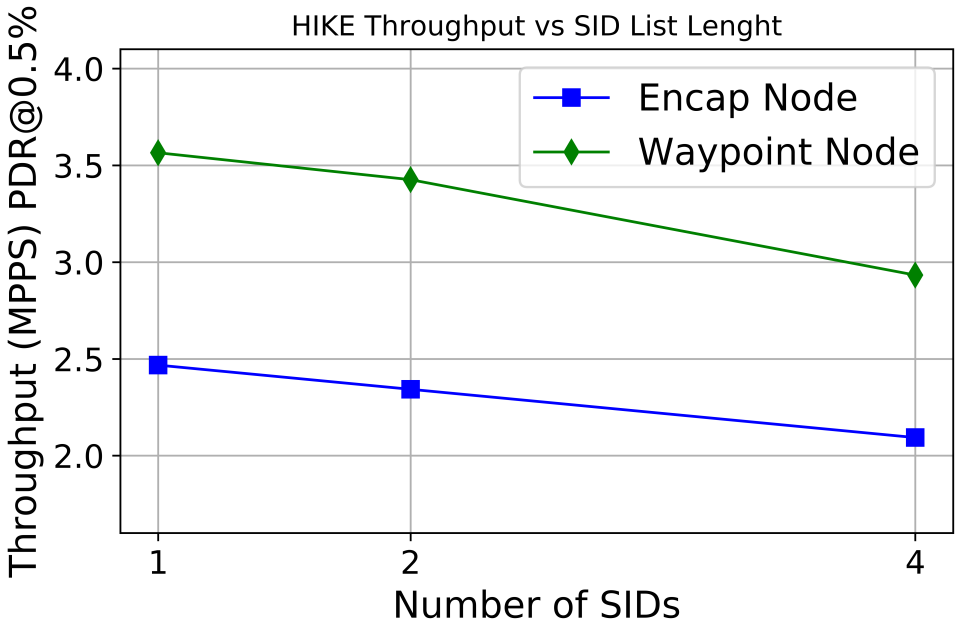


Figure 6.7: *HIKe-v0 fast path performance with respect to the number of SID in the SID List.*

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) [71] and In situ OAM (IOAM) [22], but also more complex solutions such as Alternate Marking [113]. 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 such as the "The One-Way Active Measurement Protocol (OWAMP)" [140] and the "Two-Way Active Measurement Protocol (TWAMP)" [78]. Further details about such protocols are provided in Chapter 5.

6.6.2 eBPF based solutions

In [169] authors propose eVNF, an hybrid virtual network function with Linux eXpress Data Path (XDP). In their architecture, XDP has been used for simple but critical tasks coarse packet filtering, leaving to the "slow path" (i.e. 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 [170], 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. [5] 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 [29] 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 the HIKe-

v0 implementation, I use a similar technique to control eBPF programs. However, the collection of measured data, in my case, is done using the TWAMP protocol.

In [116] it is shown how eBPF can be used to implement SRv6 instructions that are activated through lightweight tunnels in the "traditional" 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 [105], 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.

6.7 Conclusions

In this Chapter, I considered a Linux-based software router in an Hybrid SDN/SRv6 architecture exploiting the eBPF (extended Berkeley Packet Filter). Hence, I propose a data plane architecture called HIKe-v0 (HybrId Kernel/eBPF forwarding, version 0) which integrates the packet forwarding and processing based on "traditional" Linux kernel networking stack with the ones based on the eBPF VM (i.e. through XDP and TC hooks), taking the best of both worlds.

The HIKe-v0 architecture provides an organisation of eBPF programs in the SDN context, overcoming the many technical limitations given by using the XDP fast path and in the Linux Traffic Control. Thus, I implemented the architecture in a proof-of-concept, and used 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 I 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 to perform complex and customizable SDN packet processing.

CHAPTER 7

eBPF Programming Made Easy with HIKe and eCLAT

7.1 Introduction

In typical datacenter networking scenarios, the packet processing nodes are based on general purpose processors (i.e. x86) and run Linux based Operating Systems. To support the demanding requirements of workloads running on containers and VMs, there is the need for fast and efficient packet processing solutions targeted to Linux/x86 nodes [12, 83]. In the last few years, the eBPF technology has gained a prominent position among the solutions to improve the packet processing performance of such nodes [17, 62, 125, 166]. For example, eBPF is the core technology of Cilium, a leading framework for secure networking in the Kubernetes container orchestration platform [129]. Another prominent example is Katran, a layer-4 load balancer, open-sourced by Facebook [10].

While Cilium and Katran clearly demonstrate the success of eBPF, there are a few annoying limitations and issues in the eBPF architecture and development model that generate complexity, preventing a wider application of this technology and limiting the advantages that eBPF could bring [60, 111]. In particular, each eBPF program needs to be *verified* by the kernel before being loaded, this process is very annoying for the developer [118] and it can increase the development time with a significant loss of productivity. Difficulties in verifying complex and/or large eBPF programs have led developers to choose the decomposition approach. This means that the logic contained in a complex eBPF program is distributed over simpler eBPF programs that are verified individually, thus increasing the chances of passing the verification stage. However, the problem is finding a way to *chain* these programs in order to regain the business logic of the single and initial complex program.

Since the early days of eBPF, one technique for concatenating programs with each other was to exploit the *tail call* approach [19, 76]: the execution flow jumps from the *caller* eBPF program to the *callee* eBPF program with no way back and without any possibility to evaluate a return value. This approach differs from the well-known *function call* paradigm where the *callee*, once its logic is completed, returns back the control to the *caller* providing, if needed, a return value.

The support *function call* paradigm to functions/subprograms which can be verified independently of the rest of the code (of the *caller* eBPF Program) was introduced in the eBPF core recently [47]. In addition, mixing *tail call* and *function call* in the same eBPF program comes with some important restrictions: i) it was prohibited in the Linux kernel until version 5.10 and is currently only supported on x86-64 architecture; ii) the available space in the program stack decreases and the maximum number of nesting *function call* is set to 8. It may come naturally to ask why anyone would still want to use *tail calls*, but the answer is simple: performance, safety, and easiness in writing individual eBPF programs totally independent of each other.

However, despite the steps forward made by the community, there is still no abstraction of the concept of program chaining in eBPF that considers independent eBPF programs and, without changing their source code, chains them together. Currently, the only way to implement a chain of programs is to manually operate on each program belonging to the chain and explicitly add the necessary code (i.e. *tail call*) to invoke subsequent ones.

The HybrId/Kernel eBPF (HIKe-v0) framework, described in Chapter 6, proposes a way to enhance eBPF introducing an efficient and dynamic chaining mechanism that supports high speed processing and filtering pipelines. HIKe-v0 simplifies the task of the developer to dynamically add and remove eBPF processing programs/functions. This framework provides helper functions and a context to combine multiple eBPF programs for processing a packet in a similar fashion to what could be achieved in the Linux kernel leveraging Netfilter/Xtables/Iptables and custom modules. In other words, HIKe-v0 makes it possible to dynamically create *linear* chains of eBPF programs simply by specifying their references in a list and, most importantly, without having to modify the code of such programs when chains are modified.

Although HIKe-v0 allows eBPF programs to be combined in a modular fashion, the technical entry barrier for an inexperienced eBPF developer that wants to use this concatenation approach for developing complex network services is still high. In fact, in HIKe-v0 it is still necessary to deal directly with the code of eBPF programs, with eBPF maps for storing program configurations, and finally with the HIKe-v0 environment. In this Chapter, I present an enhanced and re-designed version of HIKe (version 1)[1] which, this time, stands for *Hide, Improve and desKill eBPF*. The combined "traditional" kernel networking and eBPF packet processing approaches (constituting HIKe-v0) are still at the ground of the HIKe (v1) but, in this context, the framework strives to simplify the life of non-expert eBPF users by providing powerful tools for implementing and executing complex program chains. On top of HIKe-v1, the eCLAT framework has been developed to integrate with the HIKe-v1 framework with the goal of substantial improving the applicability and usability of eBPF, lowering the barrier to develop complex eBPF based solutions.

[1] From now on, I refer to HIKe-v1 simply as HIKe.

Listing 7.1: *Example of an eCLAT* chain*. Inside a chain, eBPF programs are called as they were functions, allowing an easy and flexible programming of application logic.*

```
1 flow_rate = flowmeter(packet)
2 #drop flows greater than 10Mbps
3 if flow_rate > 10:
4     droppacket()
5 else:
6     allowpacket()
```

eCLAT (eBPF Chains Language And Toolset) is a language and toolset to dynamically compose eBPF programs. An example of an eCLAT script, which I simply call *chain*, is shown in Listing 7.1. At line 4 and 6, names refer to independently running eBPF programs which are called inside the chain as they were functions. This eCLAT chain is compiled and executed in the context of an eBPF program, but it does NOT need to be verified by the kernel. Such form of composition and execution without verification is feasible thanks to the features provided by my new HIKe framework (version 1) that offers a Virtual Machine (HIKe VM) executing program chains.

It is important to note that in the HIKe/eCLAT programming model, the final user does not need to write any eBPF program to combine already existing programs for creating processing chains. Therefore, the user does not have to go through either the quirks required to write an eBPF program nor the following compilation and verification phases.

I have structured this chapter to emphasize my contributions that are related to the conception, design, and implementation of the entire HIKe framework. In addition, I also contributed to the design of the eCLAT framework. Therefore, In Section 7.2, I discuss the type of eBPF programmability offered by HIKe/eCLAT solution and I compare it with what is offered by other eBPF frameworks. In Section 7.3, I provide some insights about the shortcomings and pitfalls of eBPF that motivate my work. I illustrate the architecture of the proposed solution in Section7.4. In Section 7.5, I outline the main and distinctive features of HIKe considering the chain-oriented programming model, the code execution, the security provided by the HIKe VM, and I also give a short description of the tools required to compile and load HIKe Chains. I dig deeper into internals of the HIKe framework, describing its main components in Section 7.6. In Section 7.7, I briefly illustrate the abstractions introduced by eCLAT and I show how this framework allows HIKe to be used in a simplified and productive way. A careful study of the performance of the entire proposed solution is carried on in Section 7.8. In Section 7.9, I present the principal contributions in the literature related to chaining programs in eBPF. Finally I draw conclusions in Section 7.10.

7.2 eBPF programmability

The development of networking code inside the kernel of an Operating System is always a task reserved to domain experts and not to generalist programmers. What typically happens is that networking tools are developed by a limited number of experts (Wizards) and then operated by a large number of users (Muggles). Take for example the Netfilter based firewall in Linux based operating systems, written by a restricted number of Wizards and operated by tens of thousands of system administrators around

the world which can use the *Iptables* abstraction to configure the firewall rules.

The eBPF development process has its nuances and issues that makes it even more difficult than normal kernel programming and restricts it to an "elite" of experts. eBPF is inherently complex, forcing the developers to program in elementary C language and sometimes even in assembly or bytecode. eBPF programming is more an art than a science [18] and the metaphor of Wizard and Muggles fits perfectly with the eBPF ecosystem. With this metaphor in mind, I can state that the goal of HIKe/eCLAT is to allow Muggles to easily *program* eBPF solutions, by writing the application logic that combines a set of eBPF programs (written by Wizards).

The implementation of new solutions based on eBPF and the evolution of the existing eBPF frameworks is constrained by the capacity of Wizards to develop inside the eBPF framework so that usable tools can be offered to Muggles. In turn, this is limited by the inherent complexity of the eBPF model and by a number of issues of the eBPF development process and toolchain. The eBPF Wizards have to fight hard against the eBPF shortcomings and pitfalls to earn their keep.

Hereafter, I briefly compare the HIKe/eCLAT solution with other existing eBPF frameworks such as Cilium [127] and Polycube [109]. The idea behind such frameworks is to employ Wizards to develop eBPF program templates (modules), leaving their final *configuration* to Muggle. All the process usually pass through a code generation engine which works in user space and is responsible for generating the final eBPF code to be injected into the kernel. The drawbacks of this approach are:

- (for Wizards) creating new modules inside such frameworks requires more effort and ability than writing conventional eBPF programs. Wizards have to be familiar with the environment where programs are going to be deployed;
- (for Muggles) the configuration does not often fully cover the need for arbitrarily complex packet processing "business logic".

Polycube offers a form of composition of its *cubes*, but this is not a composition of programs *inside* the eBPF framework. Very often, a cube is monolithic eBPF program which receives packets on a virtual interface and emits packets on another one. Composing two cubes means connecting the output interface of a cube to the input interface of another cube.

The HIKe/eCLAT approach is different, moving from *configuration* or *composition* to eBPF *programmability*. In this solution, the building blocks are made of conventional eBPF programs (referred to as HIKe eBPF Programs) which are written by Wizards and slightly adapted to work in the HIKe framework (just a few lines of code are needed). Differently from other approaches (such as Polycube's *cubes*), HIKe eBPF Programs are not necessarily complex programs (e.g., "the firewall") but can be also small utilities such as "flow meter" or "drop this packet". These HIKe eBPF Programs can be called as "functions" inside a higher level program that is called *chain*, allowing the programmability of custom business logic. A chain can be expressed using a high-level programming language called eCLAT. Hence, the chain executes the business logic through the HIKe VM which is in charge of invoking the eBPF programs (i.e. HIKe eBPF Programs) within the eBPF framework.

Using a Unix similarity, it is like having many standalone tools such as `cut`, `tail` or `sed` written by Wizards. Muggles can use them in custom bash scripts to cover a

wide range of specific application needs, together with variables definitions and constructs such as branching conditions. By providing a large number of basic programs/applications (such as the ones offered by the GNU Coreutils), a user can create very powerful scripts for configuring and programming the system. In a Unix oriented scenario, a very simple script could leverage, for instance, the *pipe* operator for *chaining* together programs whereby the the output of one program serves as the input to the next. It is worth noting that the "code" of each program belonging to a chain does not change if that chain has to be modified. eCLAT follows the Unix philosophy in which a chain is written in the eCLAT language which is similar to `sh` or `bash` and the basic programs (i.e. `cut`, `tail`, etc) are HIKe eBPF Programs provided by Wizards for being chained together into scripts. In a nutshell, eCLAT moves the eBPF programming away from C/assembly/bytecode programs towards high-level, python-like scripts. The benefits of this approach are:

- (for Wizards) developers write HIKe eBPF Programs without almost any knowledge of the HIKe framework, or they can easily adapt existing ones;
- (for Muggles) thanks to the eCLAT language, final users have a great flexibility for implementing any custom business logic, with no need to learn eBPF programming. The eCLAT scripting language has much in common with the python syntax and, thus, it is very easy to use.

Under the hood, a chain implemented through an eCLAT script gets automatically compiled by the eCLAT framework in the so-called *HIKe Chain*. An HIKe Chain is written in C language which is, in turn, compiled into an executable binary (bytecode) run by the proposed HIKe VM. The resulting HIKe/eCLAT solution is decoupled in two levels:

1. HIKe (Heal, Improve and desKill eBPF), an add-on extension to eBPF that defines a new Virtual Machine on top of the eBPF one;
2. eCLAT (eBPF Chains Language And Toolset), a high level programming framework and runtime.

HIKe is designed for the eBPF Wizards and is meant to boost the productivity of expert developers. HIKe offers the Wizards a new paradigm to compose eBPF programs, favouring modularity, code reuse and for fast prototyping. A Wizard can easily turn an eBPF program into a HIKe eBPF Program that can be reused as a component inside the HIKe framework. The proposed *HIKe Virtual Machine* (HIKe VM) is the key breakthrough on top of which HIKe and eCLAT can overcome the eBPF issues and limitations that are further discussed in Section 7.3.

eCLAT is designed for the Muggles to take advantage of the modularity and composability features of the HIKe framework without the hassles and pitfalls of the eBPF Virtual Machine, its programming in C and assembly languages. An eCLAT script is designed to arbitrarily concatenate HIKe eBPF Programs according to a specific logic, getting rid of the eBPF verifier. At the same time though, the chains are guaranteed to be safe by the HIKe VM, which is always able to detect the faulty ones and stop their execution without compromising the stability of the system.

The proposed HIKe/eCLAT solution moves from eBPF configurability/composability, to real *programmability* made easy.

7.3 A dive into eBPF shortcomings

eBPF is definitely a powerful but complex technology. Developing complex systems based on eBPF is challenging due to the intrinsic limitations of the model and the known shortcomings of thetool chain (not to mention a few bugs that can affect this toolchain). The learning curve of this technology is very steep and needs continuous coaching from experts. I already covered, without claiming to be exhaustive, the very basics of eBPF in Chapter 2, Subsection 2.3.6. In the following Subsections, I mainly focus on the shortcoming and the pitfalls of eBPF and its current implementation. This analysis provides the motivations for the HIKe/eCLAT framework that I will present in this Chapter.

7.3.1 The verification hell

The extended Berkeley Packet Filter (eBPF) [50] is a low level programming language that is executed in a Virtual Machine (VM) running in the Linux kernel. eBPF programs can be written using assembly instructions that are converted in bytecode or in a restricted C language, which is compiled using the Clang/LLVM compiler. The `bpf()` kernel system call performs a range of operations related to eBPF. It is used for loading the bytecode of an eBPF program and attaching it to a given hook. The loading usually comprises two steps: i) verification that ensures that the eBPF program is safe to run (i.e. it does not harm the system once loaded) and ii) JIT (Just In Time) compilation that translates the eBPF bytecode into the specific instruction set for a given architecture (i.e. x86, arm, 64 or 32 bits). On few architectures, JIT compilation is not available and the eBPF bytecode will be interpreted at run time once the eBPF program is executed. In this context, I am only interested in the verification phase since that is the one creating major issues (headaches :-)) in the eBPF programming model.

The kernel validation approach is almost adequate for simple eBPF programs, i.e. few instructions, loop-free code, and no complex pointer arithmetic, while it has been shown to be a very tough obstacle to the development of complex applications [111]. As analyzed in [60], there are four main issues: i) the verifier reports many false positives, forcing developers to insert redundant checks and assume quite contrived programming solutions; ii) the verifier does not scale to programs with a large number of logical paths (i.e.: nested branches); iii) it does not support programs with unbounded loops; iv) its algorithm is not formally specified. This often causes that even a semantically correct program does not pass the validation.

One of the reasons for these problems is that the compiled bytecode offered to the verification step is the results of the *optimization* procedures executed by the compiler/optimizer. For optimization reasons, the compiler/optimizer can change the sequence of operations (preserving the correctness of the program) with respect to the C source code and this can violate some constraint that must be checked by the verifier. The final bytecode obtained from the compilation of an eBPF program often depends on the version of the used compiler/optimizer (toolchain). Different versions of the compiler/optimizer may not produce identical bytecode from the standpoint of the individual instructions used as well as their order. At the same time, the eBPF verifier evolves as new features are added in the later releases of the kernel. All of this can affect the possibility that a given program compiled with a specific version of the toolchain is

correctly verified on one specific kernel version but is rejected on another. A program that is correctly loaded on a version Y of the kernel may not be supported on a version X, such that $X < Y$. The activity of backporting eBPF applications on out-of-date kernels but which are still in use (and there are several reasons for this to happen) can be a nightmare indeed.

For all the reasons above, the verification step is the main obstacle in the implementation of complex eBPF applications. Muggles who want to implement traffic processing solutions based on program chains (with remarkable performance) should stay away from the idiosyncrasies of the eBPF world but have, at the same time, the ability to concatenate existing (eBPF) programs without going insane with the verifier. To this regard, HIKe/eCLAT provide an innovative way to create chains of eBPF programs in a Python-like language, whose chains can be executed (through the HIKe VM) without being verified.

7.3.2 eBPF composability

Software engineering identifies modularity, composability and code/component reuse as the practices essential to produce quality software. Large software systems are divided into smaller components, taming their complexity. The simpler pattern to achieve this decomposition is the *function call*. With this pattern, a *caller* function hands over the control flow (and possibly providing some parameters) to a *callee* one. The *callee* function, once completed, returns the control flow (possibly with return parameters) to the *caller*.

In the early days of eBPF, the *function call* paradigm was only implemented using *inline* functions. An inline function is one for which the compiler copies the code from the function definition directly into the code of the calling function rather than creating a separate set of instructions in memory. With the *inline* approach there is no real transfer of the flow and of the parameters to a different execution context, but the program logic is "flattened out" into a single big execution context. The *inline* approach may greatly increase the size of the code block to be compiled, optimized, and verified. This may cause trouble, as the probability that the verifier will complain about the compiled and optimized code dramatically increases with the size and complexity of the source code.

In order to overcome the limitation of the inline approach, the eBPF community has recently started to introduce the "eBPF-to-eBPF call" pattern. With this pattern the called function is not inlined in the calling code, but the traditional concept of subroutine is used. This approach is only available in recent kernels and only for x86 architecture (see [129]). Moreover, there are limitations in the operations that can be executed inside the subprograms, mostly due to some issues related to the way the stack space is managed by eBPF VM.

In addition to compile-time composition[2], the eBPF core offers the possibility to compose already compiled and verified eBPF programs using the concept of *tail call*. Tail calls are a specific eBPF feature that allows an eBPF program to execute (or better, launch) another eBPF program. A *tail call* is different from a *function call* since: i) it applies to fully-fledged eBPF programs rather than subprogram/functions; ii) the execution flow does not return to the *caller* eBPF program at the end of the *callee* one. Moreover, it is not directly possible to pass parameters between the *caller* and

[2]It actually means composition by coding, compilation, and *verification* of functions/subprograms all in a single place.

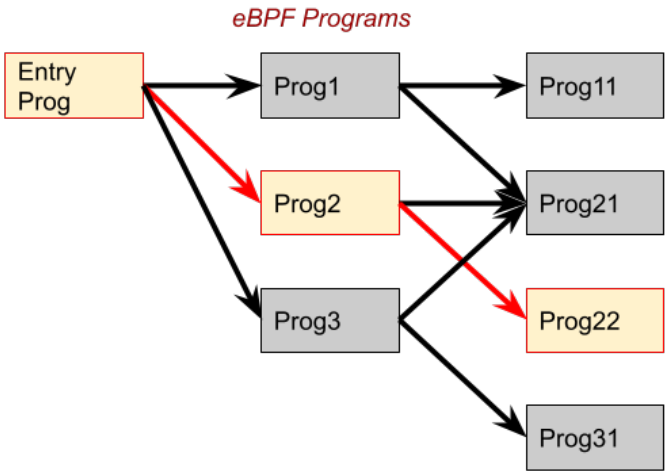


Figure 7.1: *Chaining eBPF programs with tail call based approach.*

callee eBPF programs. Once the *callee* eBPF program gets executed, it completely replaces the *caller* one; for this reason, the the eBPF VM does not allocate any extra call stack frame and it reuses the one of the *caller* eBPF program. Clearly, the developers cannot use the *tail call* according to the *function call* pattern that everyone is used to. However, *tail calls* are commonly used to implement complex systems comprising dynamic chains of programs that dispatch the packet to perform a joint elaboration [129].

To overcome the downsides of the *tail call* approach, the eBPF developers recently introduced (and still under development) the dynamic re-linking [46] of BPF programs. This approach enables an eBPF program (target) to provide a number of placeholders of global functions (extension programs) which later can be replaced with future eBPF programs. The extension program can call the same *bpf helper* functions as the target program, however it cannot recursively extends an extension. In other words, the verifier allows only one level of replacement. The dynamic re-linking is way more complex than *tail call* approach and a lot of components are involved for accomplish this new feature (i.e. BPF trampoline, function by function verification, etc). However, this approach allows parameters to be passed to extended programs (currently only the pointer to context and scalars) and enables the control flow to be returned to the target program.

7.3.3 Still lack of program chaining abstraction in eBPF framework

No matter what approach is used for linking together eBPF programs, the concept of *program chaining* is still missing in the eBPF core. A developer must always create an Entry (or *rootlet*) eBPF program (i.e. the fist program to be attached on a given hook) and then he/she must provide (by encoding) the logic used for calling the other eBPF programs[3]. Different types of function/program calling approaches (static or dynamic), in this case, are not of any help. This is because the structure of the interconnections with the next eBPF programs to be called is "statically" encoded into the *caller* one. From the perspective of developing a chain of programs, a developer needs to set up the

[3]The idea is that each eBPF program contributes with its business logic to advance package processing.

flow logic at compile time, and provide, for each program involved in the composition, all possible connections with other programs. If such a developer wants to change the structure of the interconnections, he/she needs to change the code of the involved eBPF program(s) code, re-compile and (hopefully) pass through the verification hell. For instance, this kind of strategy is implemented by the state-of-the-art eBPF composition framework called Polycube [109].

In Figure 7.1, it is possible to see an example of a composition of eBPF programs that invoke each other via *tail calls*. When an incoming packet is handed to the eBPF "Entry Prog", it can select another program to handle the packet and execute it with a *tail call*. The execution control is handed over to the *callee* eBPF program and the processing continues along all the *calling tree*.

7.3.4 The clumsiness of BPF maps

eBPF programs need to interact with user space programs to get configured and to provide information (i.e. statistics). Moreover, eBPF programs may need to access (i.e. read/write) global state information, which represents a way to exchange information among different invocations of the same eBPF programs. The eBPF framework provides the abstraction of *eBPF maps* to these purposes. The eBPF maps are key/value stores residing in the Linux kernel memory. Different types of eBFP maps are supported, i.e. with different key and value types (see [129]). The use of eBPF maps is not straightforward for the developer, in particular accessing eBPF maps from user space programs is a cumbersome operation.

Going back to the Wizard and Muggles metaphor, working directly with eBPF maps is a task for Wizards, while Muggles should be shielded from developing code to read-/write the eBPF maps. The Wizards have to design the libraries and/or GUI tools that can be used by the Muggles with basic programming skills to indirectly interact with the eBPF maps.

The risk of race conditions is another critical issue that needs to be taken into account when designing the interaction of eBPF programs and user space programs with the eBPF maps. As multiple eBPF programs can be executed in parallel to process a set of incoming packets (one per core), the access operations on the maps may conflict. For example the eBPF framework does not offer the possibility to get a lock on multiple eBPF tables to perform an atomic set of write operations. It is a task of the developer to handle the concurrency when needed and this is in general a hard task even for Wizards.

7.4 Overview of the Solution

In this Section, I discuss the solution from the point of view of a Muggle who wants to easily develop a specific packet processing procedure. The Muggle needs to write an eCLAT script and load it into the system so that it will be executed on the incoming traffic.

The architecture of the solution is shown in Figure 7.2 and it is composed of two layers: eCLAT and HIKe. In turn, the eCLAT layer is based on a user-level daemon, developed in Python and on a Command Line Interface (also developed in Python) used by the Muggle to interact with the eCLAT Daemon. The interaction between the eCLAT CLI and the Daemon is based on gRPC.

The eCLAT Daemon receives the scripts from the eCLAT Chains described in the eCLAT language. The daemon first "transpiles" them into into C language, generating the source code of HIKe Chains, then it compiles the C HIKe Chains into a executable format suitable for being loaded and executed by the HIKe VM. Actually this is not only a compilation operation, because the eCLAT Daemon also works as a *linker*: it resolves the references to HIKe eBPF Programs and to other HIKe Chains called inside a Chain and writes the HIKe eBPF Program IDs and Chain IDs into the bytecode. Moreover, the eCLAT daemon manages the dynamic compilation, verification and loading of the HIKe eBPF Programs that are referred in the Chains. In fact, when a HIKe Chain refers to a HIKe eBPF Program, the eCLAT daemon checks if that program is already loaded and if not, it loads it. The executable of a HIKe Chain (i.e. the bytecode with some additional info) is stored by the eCLAT Daemon in the HIKe Persistence Layer, which is based on eBPF maps. The eCLAT Daemon also interacts with eBPF maps in the HIKe layer, that are used by the HIKe eBPF Programs to read/write information. Further discussion on the eCLAT layer is reported in Section 7.7.

The HIKe layer provides the Runtime Environment for executing the bytecode of the HIKe Chains. As it will be detailed in Section 7.5, I have designed and implemented a Virtual Machine abstraction called HIKe VM, in the form of a "library" that needs to be included in all HIKe eBPF Programs (i.e. such programs intended to be concatenated into HIKe Chains, at least). Therefore, to run an eBPF program in the HIKe framework, the source code of the program needs to be slightly modified by adding the calls to the HIKe VM library and the program needs to be recompiled/verified/loaded as any eBPF program.

7.4.1 An eCLAT Script example

As an example, I consider the following packet processing logic that a Muggle wants to implement using eCLAT.

> If the packet rate for any IP destination D is over a threshold R1, analyze all IP sources S_{any} that are sending packet for this "overloaded" IP destination D.
>
> If an IP source S in S_{any} is sending packets with a packet rate over a threshold R2, put the IP (S, D) in a blacklist for a duration of T seconds.
>
> During this interval T, drop all packets in the blacklisted (S, D) couple and send a sample of the dropped packets (e.g., one packet every 500 packets) to a collector.

The logic refers to a DDOS mitigation scenario, implementing this logic for a non-skilled eBPF programmer is not easy. In eCLAT, the non experienced programmer (Muggle) can write a script like the one shown in Listing 7.2. Specifically, the script `ddos_tb_2_lev` (DDOS with two levels token bucket) uses and combines in a custom way 7 different eBPF HIKe Programs, which are imported in lines 1-3. The HIKe Chain Loader is called `ip6_sc` (line 4) and it selects all the IPv6 packets. The HIKe Chain Loader/Traffic Classifier is configured in line 10, which binds the HIKe Chain `ddos_tb_2_lev` to the loader. The HIKe Chain Loader/Traffic Classifier will be investigated later, but for now it suffices to say that this is the program directly attached to the XDP hook. Consequently, for each incoming packet it determines (based on its own logic) which HIKe Chain should be executed on the packet. At line 11 the `ip6_sc` is attached to the XDP hook of `eth0` interface.

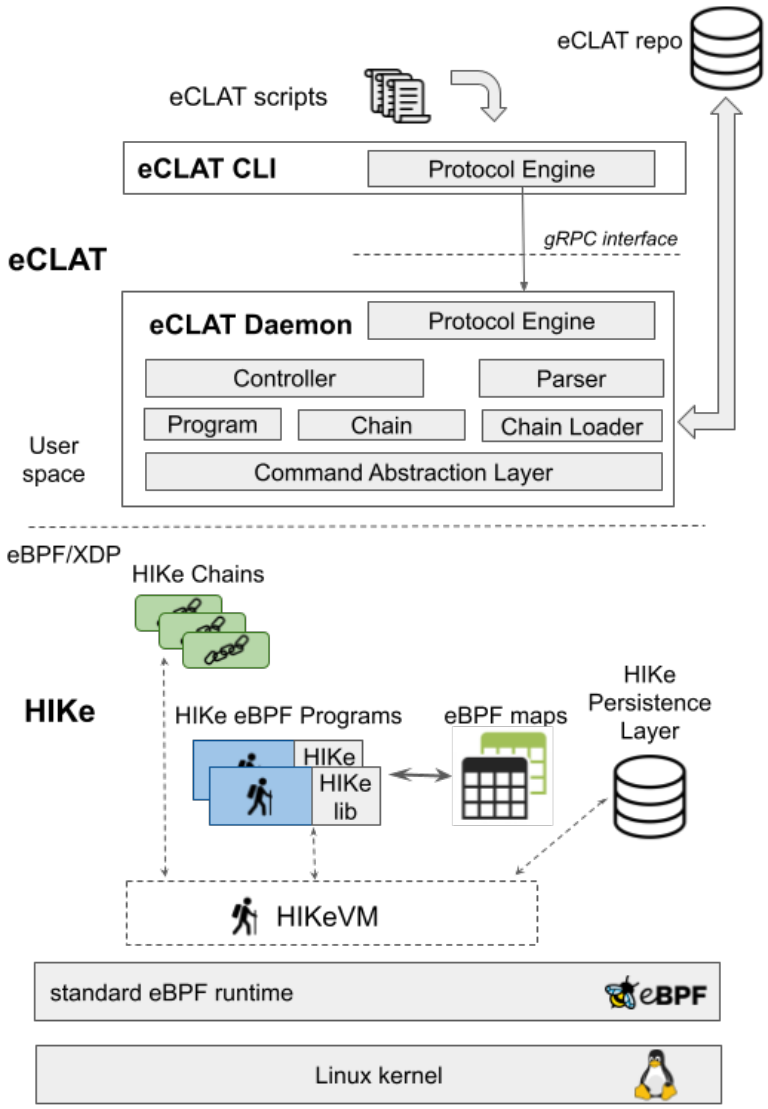


Figure 7.2: *HIKe/eCLAT overall architecture.*

The logic of the `ddos_tb_2_lev` chain is defined starting from line 13, as follows.

1 call the `ip6_hset_srcdst` program with parameter (`LOOKUP`). The result is 0 if the IPv6 (src, dst) is blacklisted;

2 if the packet is blacklisted, send one packet every 500 to an interface that collect packet samples and drop the others (line 14); count the REDIRECT and the DROP events;

3.a if the packet is not blacklisted, check the IPv6 destination against a token bucket. If the rate is out of profile for the token bucket (per destination), check the IPv6 (source, destination) against another token bucket. If the rate of the (source, destination) flow is out of profile, put the (source, destination) flow in the blacklist by calling again the `ip6_hset_srcdst`, this time with parameter ADD, and then

drop the packet (increasing the DROP events counter);

3.b if the packet is not out of the profile, increment a counter of the passed packets and exit the eBPF program by handing the packet to the regular kernel processing.

eCLAT scripts support branching and looping instructions (if, for, while, although in the limits set by the HIKe VM and eBPF verifier), and simplify the operations to read-/write packets (resolving the endianess automatically). Variables are typed, using the Python syntax for Syntax for Variable Annotations (PEP 526) [63]. The data returned by HIKe Chains and HIKe eBPF Programs are 64 bit long but can be cast to shorter subtypes.

What is evident from this (simple) example script, eCLAT provides the flexibility to define custom application logic in an easy way, by reusing different standalone HIKe eBPF Programs as they were Python functions. It is worth noting again that this eCLAT script does not have to go through the eBPF verifier, even though it uses HIKe eBPF Programs triggered by the script logic to process packets.

Listing 7.2: *eCLAT script for DDOS mitigation.*

```
1  from prog.net import hike_drop, hike_pass,  \
2       ip6_hset_srcdst, ip6_sd_tbmon, monitor,\
3       ip6_dst_tbmon, ip6_sd_dec2zero, l2_redirect
4  from loaders.basic import ip6_sc
5
6  # send all IPv6 packets to our chain
7  ip6_sc[ipv6_sc_map] = { (0): (ddos_tb_2_lev) }
8  ip6_sc.attach('DEVNAME', 'xdp')
9
10 def ddos_tb_2_lev():
11     PASS=0; DROP=1; REDIRECT=2;
12     REDIRECT_IF_INDEX = 6;
13     ADD=1; LOOKUP=2;
14     BLACKLISTED = 0;
15     IN_PROFILE = 0;
16
17     # (src,dest) in blacklist ?
18     u64 : res = ip6_hset_srcdst(LOOKUP)
19     if res == BLACKLISTED:
20         # redirect one packet out of 500
21         res = ip6_sd_dec2zero(500)
22         if res == 0:
23             monitor(REDIRECT)
24             l2_redirect(REDIRECT_IF_INDEX)
25             return 0
26
27         monitor(DROP)
28         hike_drop()
29         return 0
30
31     # check the rate per (dst)
32     res = ip6_dst_tbmon()
33     if res != IN_PROFILE:
34         # check the rate per (src,dst)
35         res = ip6_sd_tbmon()
36         if res != IN_PROFILE:
37             # add (src,dest) to blacklist
38             ip6_hset_srcdst(ADD)
39             monitor(DROP)
40             hike_drop()
41             return 0
42
43     monitor(PASS)
44     hike_pass()
```

```
45      return 0
```

7.5 HIKe: Heal, Improve and desKill eBPF

The shortcomings of the eBPF, its development process and toolchain motivates my work. My goal is to compose eBPF programs using the *function call* pattern and without the hassles of the verification phase. Unfortunately, these requirements cannot be met by the current eBPF framework. Furthermore, it is not possible to directly enhance eBPF to meet such requirements as it would violate some fundamental assumptions (i.e. the safety guarantees provided by the verifier).

My proposed approach is based on a new lightweight Virtual Machine abstraction (HIKe VM) running on top of the existing eBPF VM. The HIKe VM is a register-based Virtual Machine using a subset of the eBPF VM 64-bit RISC instruction set. Through the HIKe VM, it is possible to execute HIKe Chains which combine eBPF programs (actually HIKe eBPF Programs) leveraging the familiar and well-known *function call* paradigm. A HIKe Chain is represented through a bytecode (i.e. executable instructions) and some additional info such as the number of valid instructions, the HIKe Chain Identifier (Chain ID), the version, etc. This bytecode is interpreted by HIKe VM which fetches, decodes and executes the instructions encoded inside. The bytecode codifies logical and arithmetical instructions, jump instructions to control the program flow, calls to HIKe eBPF Programs and to other HIKe Chains.

Technically, a HIKe eBPF Program is an eBPF program that could be invoked in a HIKe Chain. As will be explained in detail in Section 7.6.2, a HIKe eBPF Program is composed of two parts: i) the Program Logic that contains the custom operations to be applied to the packet being processed (implemented by the Wizard); ii) the HIKe VM code imported as a library that enables the execution of a HIKe Chain bytecode aiming to concatenate this program with the other ones in the chain. Consequently, when a HIke eBPF Program is executed (by the eBPF VM), the Program/Business Logic is always executed first and then the HIKe VM code.

In Figure 7.3, I illustrate the interactions among the key components of HIKe, such as: HIKe Chain, HIke eBPF Program and the HIKe VM. As can be noted, the HIKe Chain *mychain1* is executed by the HIKe VM embedded in each HIKe eBPF Program that is part of that chain. This aspect is crucial: each HIKe eBPF Program has a custom Program Logic since it can process the packet differently from another, however the code of the HIKe VM is the same for every possible HIKe eBPF Program. Likewise, it is evident that the HIKe VM is always executed after completing the Program Logic of the specific program. A step-by-step example of a HIKe Chain run by the HIKe VM is discussed in Subsection 7.5.2.

Since the bytecode of a HIKe Chain is executed by the HIKe VM, it does not need to be verified in any way. If a HIKe Chain executes invalid actions or performs an illegal operation, the HIKe VM stops the execution of that chain and takes a default decision on the fate of the packet (currently, the packet is dropped). Misbehaving HIKe Chains are reported in logs which are accessible through the Linux *tracefs* file system [104] or the eCLAT CLI.

The HIKe virtual machine, helper function libraries, and eBPF maps belonging to the HIKe Persistence Layer compose, together, the HIKe Runtime Environment (HIKe

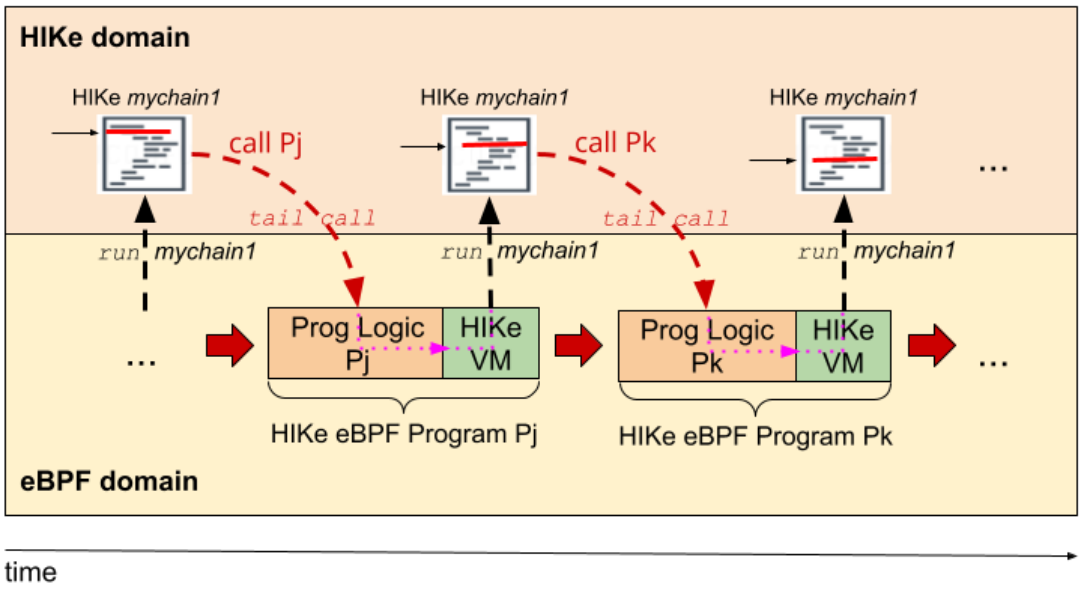


Figure 7.3: *A high-level graphical representation of HIKe Chain processing by the HIKe VM embedded in the HIKe eBPF Programs.*

RE). The HIKe RE along with the HIKe Chain Loader/Traffic Classifier, the HIKe eBPF Programs and the HIKe Chains form the so-called HIKe framework. The purpose of components not yet covered will be addressed in later Sections.

7.5.1 HIKe program chaining and its advantages

To identify the key aspects underlying the program chaining in HIKe and its advantages, it is worthwhile to give an example of how a chain of programs can be implemented in the eBPF traditional way and using the HIKe framework. For this purpose, I reported in Figure 7.4 an example chain implemented: a) using the traditional eBPF approach based on the use of *tail calls*[4]; b) using a HIKe Chain executed through the HIKe VM.

In case (a), the logic determining the next program to be executed is embedded within the programs of the same chain. So, these programs are designed to encode the logic needed to transfer control (i.e. using *tail calls*) to the next program as well as find expedients for the passage of any execution context (i.e. status on the processing carried on up to that point). Consequently, eBPF programs belonging to this chain cannot be generically used in a new chain that has a different pattern of interaction among its forming programs.

In case (b), the chain logic is represented through a HIKe Chain (described through the pseudo-code, in this example) which allows the HIKe eBPF Programs to be invoked. HIKe eBPF Programs do not have to contain any logic needed to invoke the next programs in the chain. The HIKe VM is responsible for executing the instructions contained in the HIKe Chain and invoking the HIKe eBPF Programs appropriately. As a result, the HIKe VM requires the eBPF VM to run the HIKe eBPF Programs on its behalf. Upon completion of each called HIKe eBPF Program, the flow control returns

[4]To chain eBPF programs together I consider the *tail call* based approach since, at the time of this writing, it is still the most widely adopted in existing framework such as Poliycube [109].

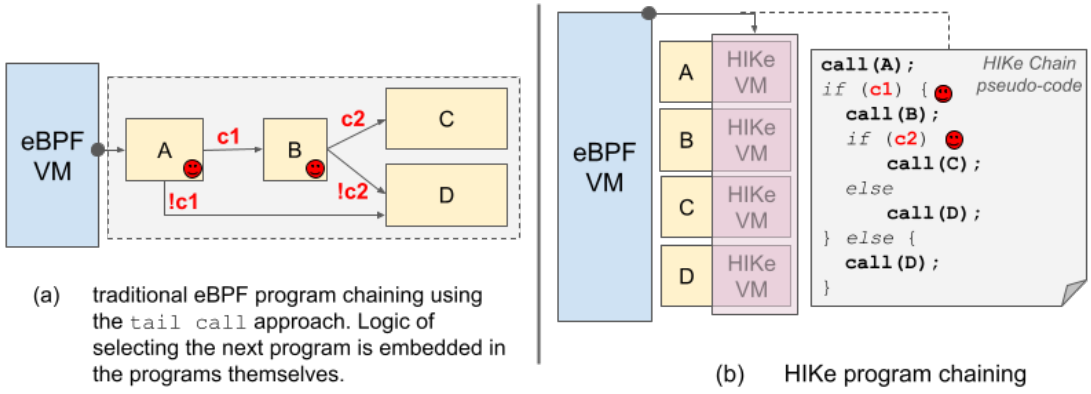


Figure 7.4: *Comparison between two different program chaining approaches: (a) traditional eBPF approach based on tail call where the logic of selecting the next eBPF program is embedded in the programs themselves; (b) HIKe Chain approach where the logic for calling the (HIKe) eBPF Programs is completely moved into the Chain run by the HIKe VM.*

to the HIKe VM, which continues with the execution of the HIKe Chain.

By comparing solutions (a) and (b), it can be observed that: i) HIKe eBPF Programs that are composed are not aware of the structure of the interconnections; ii) the HIKe eBPF Programs are compiled and verified once for ever; iii) the HIKe Chain flow logic can rely on the results of the execution of the HIKe eBPF Programs, but changing the HIKe Chain logic is NOT subject to the eBPF verifier: a HIKe Chain is compiled for the HIKe VM, not for the eBPF VM!

Considering Figure 7.4, it is straightforward to see how easily and quickly a chain can be implemented in HIKe. Rapid prototyping through HIKe is an activity that allows the developer to focus on the business logic (i.e. Program Logic) of the HIKe eBPF Programs rather than spending time, as in the traditional eBPF approach, coding from scratch the way in which programs are to be invoked.

The capabilities of the HIKe framework extend beyond just chaining of programs. Indeed, the HIKe VM allows a HIKe Chain to invoke other HIKe Chains. The invocation of a HIKe Chain uses the same *function call* pattern used to compose HIKe eBPF Programs, maintaining the same syntax and semantics. The HIKe VM, therefore, provides a unified and simple calling model for invoking both other HIKe Chains and HIKe eBPF Programs, promoting and maximizing the code reuse.

Finally, I would like to mention another breakthrough of the HIKe framework. The application domain of the HIKe VM concerns network packet processing and a HIKe Chain, describing the overall processing to be applied to a packet, is *associated* to the packet itself. Thus, a HIKe Chain "follows" the packet through its "journey" across the different HIKe eBPF Programs and Chains that are executed. This avoids the "reclassification" issue that affects the normal function chaining approaches, which often require to keep some state information to reclassify packets after a function has been executed to forward the packet to the next function. In other words, a HIKe Chain keeps track of some useful information about the packet as it is being processed, i.e. the types of network protocols currently being analyzed. In addition, a HIKe Chain can directly access the received packet and read/write its content without necessarily deferring this operation to the application logic of an HIKe eBPF Program to be called.

7.5.2 A step-by-step HIKe Chain execution example

As already pointed out, the HIKe VM is included as a library inside all the HIKe eBPF Programs. By observing Figure 7.5, the processing of a packet by HIKe starts when the packet is intercepted by the XDP hook and handed to a special HIKe eBPF Program that has the task to associate a *root*[5] HIKe Chain to the packet (C1 in the example), based on some classification rules. This peculiar program is generally referred to as HIKe Chain Loader but, because of classification rules, the more meaningful name of Traffic Classifier was chosen as reported in Figure 7.5. After executing its classification logic and identifying the chain to be executed, the Classifier starts the *boostrap* procedure. This procedure marks the HIKe Chain C1 as the *current active chain* by pushing its reference into the HIKe VM Chain Stack. This stack allows the HIKe VM to keep track of the chain being executed as well as to support nested calls to other HIKe Chains. Once the *boostrap* procedure is terminated, the bytecode of HIKe Chain C1 starts to be executed thanks to the HIKe VM code embedded into the eBPF Traffic Classifier.

As can be noted from Figure 7.5, when the current instruction of the C1 corresponds to a *function call* to HIKe eBPF Program P1, a *tail call* is executed and the packet is processed according to the program logic of P1. After the execution of P1 program logic (written by the developer to process packets), the control flow is virtually moved again to the HIKe Chain C1: this means that program P1 is executing the HIKe VM code which is included in P1 as well. It is important to note that the HIKe VM, executed in the context of a given HIKe eBPF Program, can resume the chain to be processed since it is pointed by the top of the HIKe VM Chain Stack. When the instruction of C1 requests the execution of another HIKe Chain C2 through a *function call*, this is handled by the HIKe VM, with no need to change the running HIKe eBPF Program. Indeed, the HIKe VM suspends the execution of the HIKe Chain C1 and marks C2 as the active running chain by pushing its reference on the top of the HIKe VM Chain Stack (carrying out a full chain context switch). At this point, the chain C2 which received the value of x as input parameter is run by the HIke VM. It is worth noting that C2 is still executed on behalf of the HIKe VM code embedded into the P1 program.

When the HIKe Chain C2 needs to invoke the HIKe eBPF Program P2, a second *tail call* is performed. The program logic of P2 is first executed by having access to the value of x provided as input parameter, and then the control hands over to the HIKe VM that resumes and runs C2 in the context of P2. Once the chain C2 ends, its reference is removed from the top of the HIKe VM Chain Stack and the execution flow returns to the *caller* HIKe Chain C1, still executed by the HIKe VM in the context of program P2. Chain C1 retrieves the returned value from C2 which is then used as the input argument when P3 gets called.

The execution of the *root* HIKe Chain ends as soon as an instruction calls a HIKe eBPF Program that explicitly decides the fate of the packet: pass to kernel, drop, send it over the network through the NIC. Such a program is generally referred to as *final* HIKe eBPF Program. In this example, the HIKe Chain C1 invokes the *final* HIKe eBPF Program P3 that terminates the chain execution passing the packet to the kernel networking stack for further processing. If a *root* HIKe Chain does not terminate invoking a *final* program, the fate of the packet is decided by the HIKe VM according to a default option, which in this case corresponds to drop that packet.

[5] With the term *root* HIKe Chain I refer to the first HIKe Chain associated to the network packet.

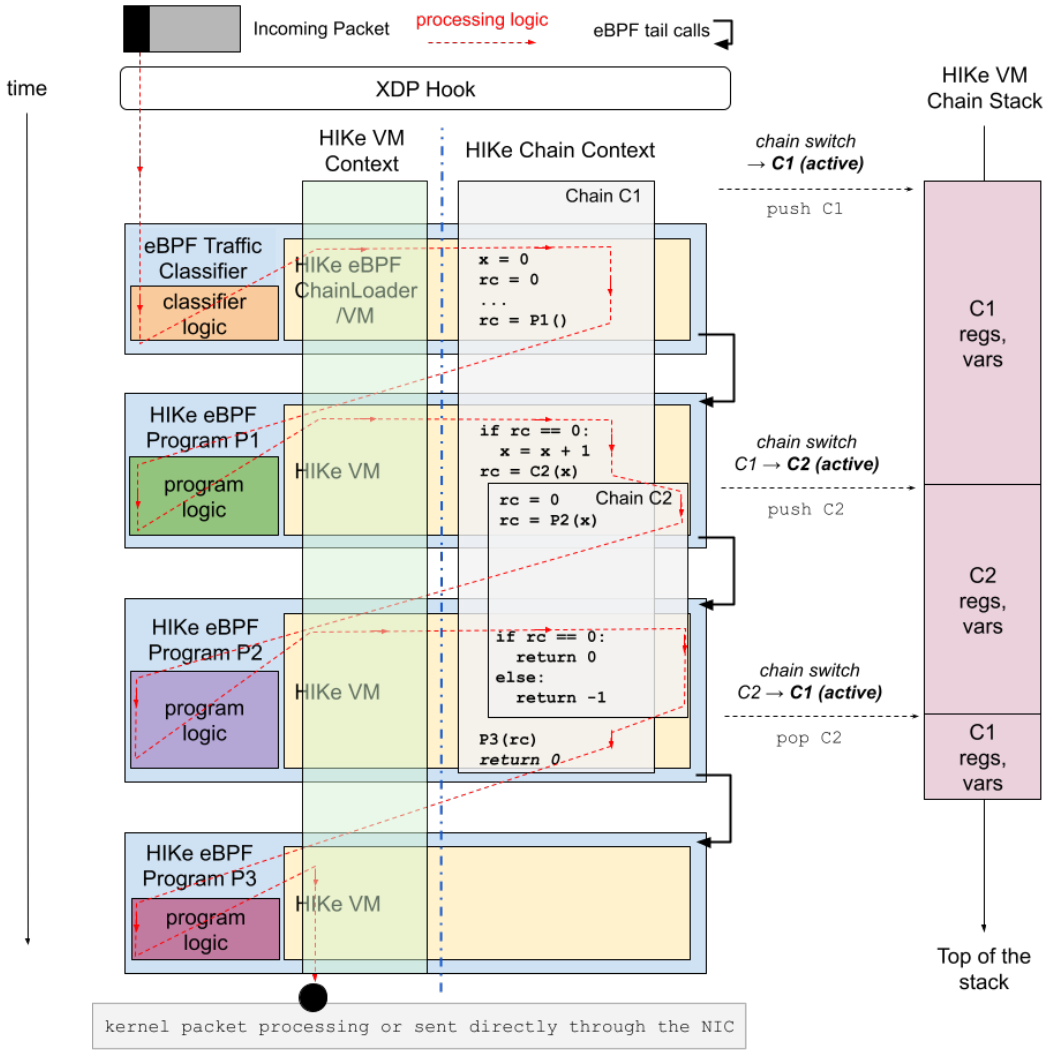


Figure 7.5: *Composition of HIKe eBPF Programs through a HIKe Chain run by the HIKe VM (runtime view).*

The memory footprint of the HIKE VM library that needs to be included in each HIKe eBPF Program is in the order of 1K instructions. In addition, it does not affect the verification phase, as the root user can verify and load eBPF programs with up to 1M instructions [48].

Writing HIKe eBPF Programs is straightforward for an eBPF programmer (Wizard). To turn an eBPF program into a HIKe eBPF Program, it has to be "decorated" with some helper functions contained in the HIKe library. In practical terms, this means adding 3 or 4 lines of C to the source code as I will describe in Subsection 7.6.2. Then the HIKe eBPF Program is recompiled, verified, and loaded as any eBPF program.

Deeper technical details on the design and implementation of HIKe VM and the Runtime Environment are reported in the next Sections.

7.5.3 Safety of HIKe VM and Chains

While every single HIKe eBPF Program needs to be compiled and verified, the bytecode of a HIKe Chain does not need to pass through the eBPF verifier. Nevertheless, the HIKe VM guarantees to meet all the eBPF safety constraints. From the point of view of the eBPF framework, the HIKe VM is seen as a normal eBPF program running in the XDP context. Since every instruction of a HIKe Chain is executed by the HIKe VM, which is in turn an eBPF program, the verifier makes sure that the HIKe VM cannot be harmful for the system. The key idea is that the HIKe VM code executor/interpreter considers every possible fault/exception raised by an implemented instruction and it acts accordingly. Thus, in case of illegal operations (i.e.: de-referencing a NULL pointer, unbounded loops, etc), the HIKe VM aborts the execution of the faulty chain. As a consequence, HIKe Chains are safe by construction and do not force the developer to adopt an unnatural code programming style with the tricks needed in traditional eBPF programs in order to be considered valid.

Summing up, the bytecode of a HIKe Chain is interpreted by the HIKe VM which is embedded in an HIKe eBPF program that has been previously verified before being loaded. Consequently, the HIKe VM interprets each instruction (bytecode) of the chain and translates it into eBPF operations that are guaranteed to be safe by the verifier.

7.5.4 Toolchain for compiling and loading HIKe Chains

I designed and implemented the HIKe VM to reuse a (large) subset of the eBPF VM instruction set, hence its bytecode[6] is piratically compatible with the eBPF VM. Furthermore, both eBPF VM and HIKe VM leverage the same Application Binary Interface (ABI) enabling the latter to take advantage of a stable and well-defined interface for called code (described by calling conventions) as well as type representation. Thanks to this design choice, a Muggle can write a HIKe Chain in C language and compile it using the Clang/LLVM toolchain for generating an ELF file object. This file contains several segments referring to one or more sections, among them is the `text` segment which contains the executable instructions of the HIKe Chain. I realized a tool that extracts such instructions from the file object, adds a header and creates the overall bytecode that forms the HIKe Chain, runnable on the HIKe VM. Hence, the generated bytecode is made of two mains sections: i) a header structured in a series of fields ii) and the executable instructions which encode the logic of the chain. Some fields of the header contain information regarding the version and identifier of the chain, the number of instructions present in the bytecode, while other fields are intended for internal use by the HIKe VM.

To simplify the whole process of compiling a HIKe Chain, I created an additional tool that automatically performs all the operations described above. The tool is called `hikecc` and takes as input the ".hike.c" file containing the definition of one or more HIKe Chains and produces a bash ".sh" script that embeds the bytecode code for each HIKe Chain. The generated bash script also contains the code needed to load the code of the HIKe Chains into the HIKe Runtime Environment.

The *hikecc* tool is heavily used by eCLAT to automate the compilation of the chains, for generating the binaries to be loaded into the runtime. At the same time, eCLAT also

[6]The Instruction Set Architecture of HIKe is compatible with the one of the eBPF VM. At the time of this writing, only atomic instructions are not implemented by the HIKe VM yet.

handles a whole range of internal operations such as retrieving information about the HIKe eBPF Programs that are invoked by the HIKe Chains, downloading them from local/remote repositories, etc.

The HIKe framework fully benefits from the maturity of the Clang/LLVM toolset and a thin adaption layer for creating bytecode of HIKe Chains to be run by the HIKe VM, with no need to reinvent each time the (very complex) wheel, i.e. compilers, optimizers, and so on.

7.6 HIKe deep dive

In this Section, I elaborate on the internals and operations of the HIKe Virtual Machine (HIKe VM) and the HIKe Runtime Environment (HIKe RE). In Subsection 7.6.1, I detail on the Instruction Set Architecture (ISA) supported by the HIKe VM, the instruction execution mode, calling conventions and then I explain the adopted memory model/-management mechanisms. In Subsection 7.6.2, I examine how a HIKe eBPF Program is constructed and how it interfaces with both HIKe VM, eBPF VM and the Runtime Environments. A similar discussion applies to HIKe Chains, which are described in Subsection 7.6.3. In Subsection 7.6.4, I discuss about the HIKe Chain Loader within the HIKe RE, while in Subsection 7.6.5 I clarify the role of the HIKe Persistence Layer. The HIKe development process is described in Subsection 7.6.6. Finally, I investigate the valuable portability properties of the HIKe Chains bytecode in Subsection 7.6.7.

7.6.1 The HIKe Virtual Machine

The HIKe Virtual Machine is a RISC register-based machine that shares (almost) the same[7] Instruction Set Architecture (ISA) of the eBPF Virtual Machine. The HIKe VM also adopts the same calling conventions and types of the eBPF VM. The HIKe VM relies on a total of 11 64-bit registers, a program counter and a fixed-size stack. Such VM is designed to execute the bytecode of a HIKe Chain which has been compiled with the toolchain provided with the HIKe framework (see Subsection 7.5.4). However, the HIKe VM has to be considered a *restricted* eBPF VM since it is not capable to execute eBPF helper functions (unless exported opportunely) or get access to generic eBPF maps directly.

In Figure 7.6, I have reported the internal structure of the HIKe VM and the main components of the HIKe Runtime Environment to which the VM interacts with during the execution of a HIKe Chain. When the HIKe VM is started, the Executor accesses the top of the HIKe VM Chain Stack which has been initialized by the HIKe Chain Loader/Traffic Classifier pushing the *execution context* of the chain to be run. The execution context of a HIKe chain contains the ID that uniquely identifies the chain in the HIKe Runtime Environment, the snapshot of the HIKe VM registers, filled/spilled registers, the values of the local variables assumed during the last execution of that chain and some ancillary data. Thus, the HIKe VM keeps the internal values of its registers up-to-date with those of the chain execution context. In addition, the VM retrieves the bytecode of the chain to be run from the HIKe Persistence Layer (specifically, from the HIKe Chain Table). At this point, everything is ready for the Executor to start fetching, decoding, and executing the instructions provided through the bytecode.

[7] At the moment, only *atomic operations* are not implemented yet.

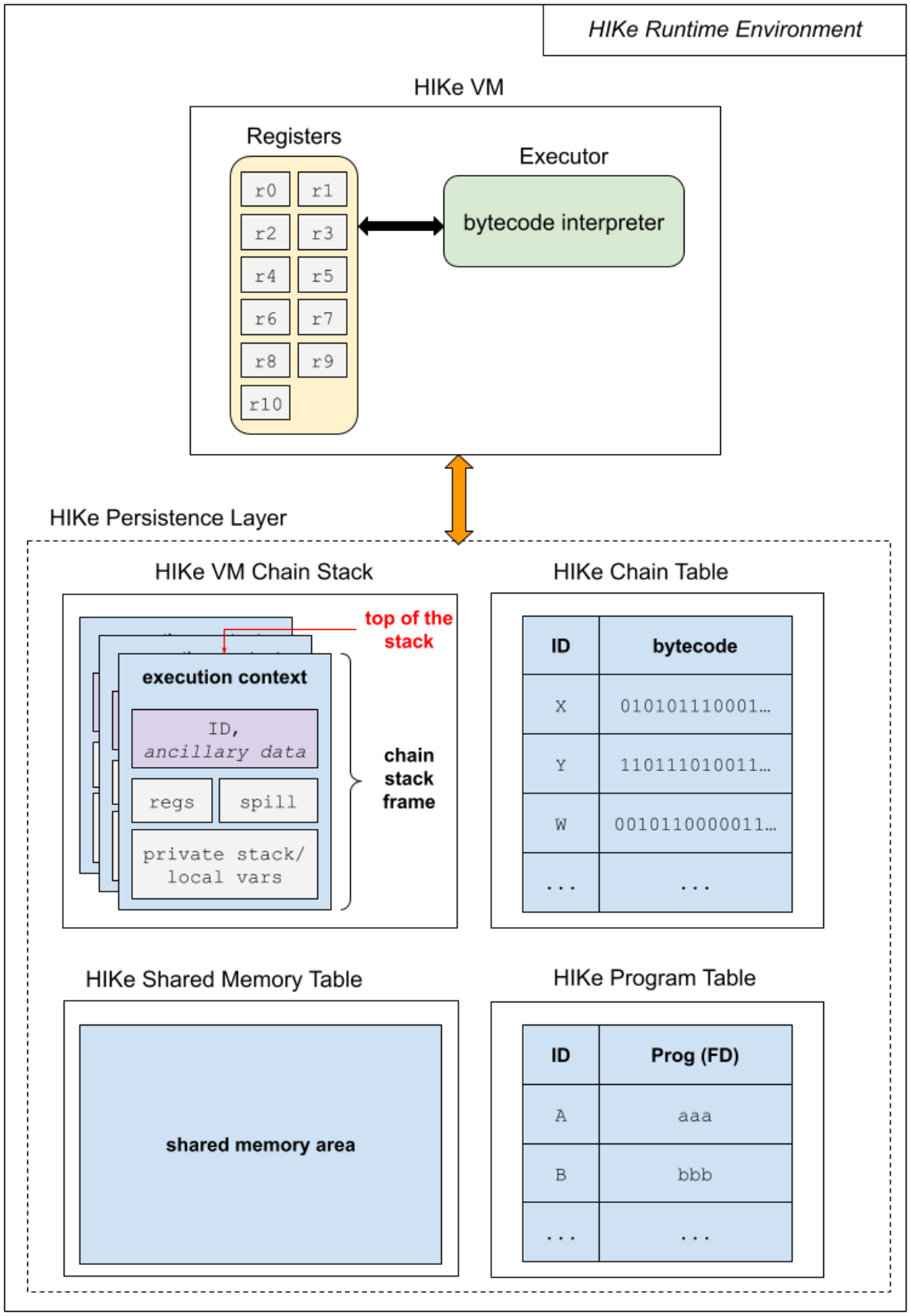


Figure 7.6: *Overview of the HIKe Runtime Environment.*

The HIKe VM does not withstand by itself and it must be embedded into eBPF programs run by the eBPF VM. For this reason, I have supplied the HIKe VM code as a library that must be imported into every eBPF program to be used in a HIKe Chain. As already explained, such programs are named HIKe eBPF Programs, indicating that they contain the code needed to run the HIKe VM. Every HIKe eBPF Program is identified with an ID that is used both from the eBPF and the HIKe REs.

Using the HIKe VM and the *function call* pattern to compose HIKe eBPF Programs and HIKe Chains, it may seem that one can overcome the intrinsic limitations imposed by the eBPF VM and provide a Turing complete execution environment. This is NOT the case, otherwise I would have found a way to overcome the security constraints of the eBPF environment. There are several limitation of the HIKe framework that must be taken into account: i) a static limitation in the number of instructions that compose the bytecode of a HIKe Chain; ii) a dynamic limitation on the number of instructions that can be executed by an instance of a chain. Both static and dynamic limitations can be changed by recompiling the HIKe VM, I have set them both to 64. The dynamic limit affects the verification phase when the HIKe VM is compiled and verified as any eBPF program. The higher the limit, the longer the time the verifier will take to check the validity of the HIKe VM before loading it.

HIKe VM Instruction Set and helper functions

The HIKe VM has 10 general purpose registers and a read-only frame pointer register (in total, 11 registers), all of which are 64-bit wide. The HIKe VM adopts the same calling conventions and types of the eBPF VM and the VM registers are intended to be used as follows:

- `r0`: contains the return value of a *function call* to a HIKe eBPF Program/Chain;
- `r1-r5`: hold arguments to be passed to a HIKe eBPF Program/Chain;
- `r6-r9`: *callee* saved registers preserved on HIKe eBPF Program/Chain call;
- `r10`: read-only frame pointer to access the HIKe VM Chain Stack.

`r0-r5` are scratch registers and HIKe Chains use to fill/spill them if necessary during a *function call* to a HIKe eBPF Program/Chain or when the pressure on registers is high. The `r10` always points to the frame held in the HIKe VM Chain Stack, containing saved and spilled registers and local variables of the executing chain.

The instruction set supported by the HIKe VM is a (large) subset of the one implemented by the eBPF VM and, thus, the instruction encoding is the same. The instructions are classified into *classes* that correspond to operations such as *load/store*, *arithmetic/logic* (64-bit), *jump* (64-bit) and so on. The HIKe VM supports all the *classes* of eBPF VM, except for the 32-bit ones. For *load/store classes*, the HIKe VM does not support *atomic* instructions yet. The list of supported instructions by the HIKe VM is available at [107], while [49] reports a in-depth explanation about all the details regarding the instruction encoding I adopted also for my VM.

HIKe does not support the same helper functions available for the eBPF. HIKe VM/RE is meant to work on top of the eBPF/XDP and, for these reasons, I provided only those features which are meaningful for packet processing scenarios. Thus, the

HIKe VM implements few helper functions which are used for accessing the packet in read/write mode directly. It is worth pointing out that the HIKe VM implements the *function call* feature through helper functions.

HIKe VM and Execution Mode

When a HIKe Chain is executed, it can run in different modes: the *chain mode* or *vm mode*. It runs in the *chain mode* when the HIKe VM executes operations/instructions of the chain which do not require a privilege such as arithmetic/logic instructions, jump to address, conditional jumps, etc. However, when the HIKe Chain has to execute a *function call* to a HIKe eBPF Program/Chain, the execution mode changes from the *chain mode* to *vm mode* and the HIKe VM takes over the control. Calling a HIKe eBPF Program/Chain from a HIKe Chain is a privileged operation, since it does not only affect the execution of the *caller* chain but also requires coordination between the HIKe VM and the HIKe RE, involving many state changes for both.

Figure 7.7 shows how a HIKe Chain running a privileged operation/instruction, such as *function call* to a HIKe eBPF Program, alters the *execution mode* causing the chain suspension. Looking at the example, the *function call* to a HIKe eBPF Program is requested by the chain C1 at the instruction $k + 1$ (step 1). The HIKe VM starts to run the instruction and the processing takes place in several stages: i) it moves ahead the HIKe Chain Program Counter (PC); ii) it decodes the instruction and accesses the VM register `r1` containing the ID of the HIKe eBPF Program P2 to be called; iii) it invokes the `bpf_tail_call()` for executing the P2 (step 2). At this point, the chain C1 is suspended as well as the HIKe VM run by the program P1. The eBPF VM runs the HIKe eBPF program P2 starting from the code that implements the business logic. Once this code has been fully executed, the control flow moves to the HIKe VM code (step 3). Here, the HIKe VM is able to determine which HIKe Chain was suspended and restores its execution context. Then, the HIKe VM returns to execute the bytecode of chain C1 starting from instruction at PC equals to $k + 2$, in *chain mode* (step 4).

HIKe VM Memory Management

In order to execute the HIKe Chains associated with the packets being processed, the HIKe VM implements memory management mechanisms that make it possible to: i) isolate the execution contexts of the HIKe Chains; ii) support the *function call* pattern typical of imperative programming; iii) provide transparent access to the bytes of a packet for read/write operations; iv) provide the *Shared Memory Area* (SMA) through which eBPF programs, HIKe eBPF Programs, and HIKe Chains can exchange information.

For performance reasons, the HIKe VM maintains the information about the running HIKe Chain separately for each logical CPU (i.e. for each core). In particular, the HIKe VM keeps the reference to the frame of the chain stored in the HIKe VM Chain Stack. Such a frame contains spilled/filled registers and local variables; it is leveraged for supporting the *function call* to another HIKe Chain as it stores the execution context of the *caller* chain. Such an organization of memory enables the HIKe Chains to be independent and isolated from each other.

The Shared Memory Area (SMA) is a scratch memory area available in the HIKe Runtime Environment that could be used by HIKe Chains, eBPF Programs and HIKe

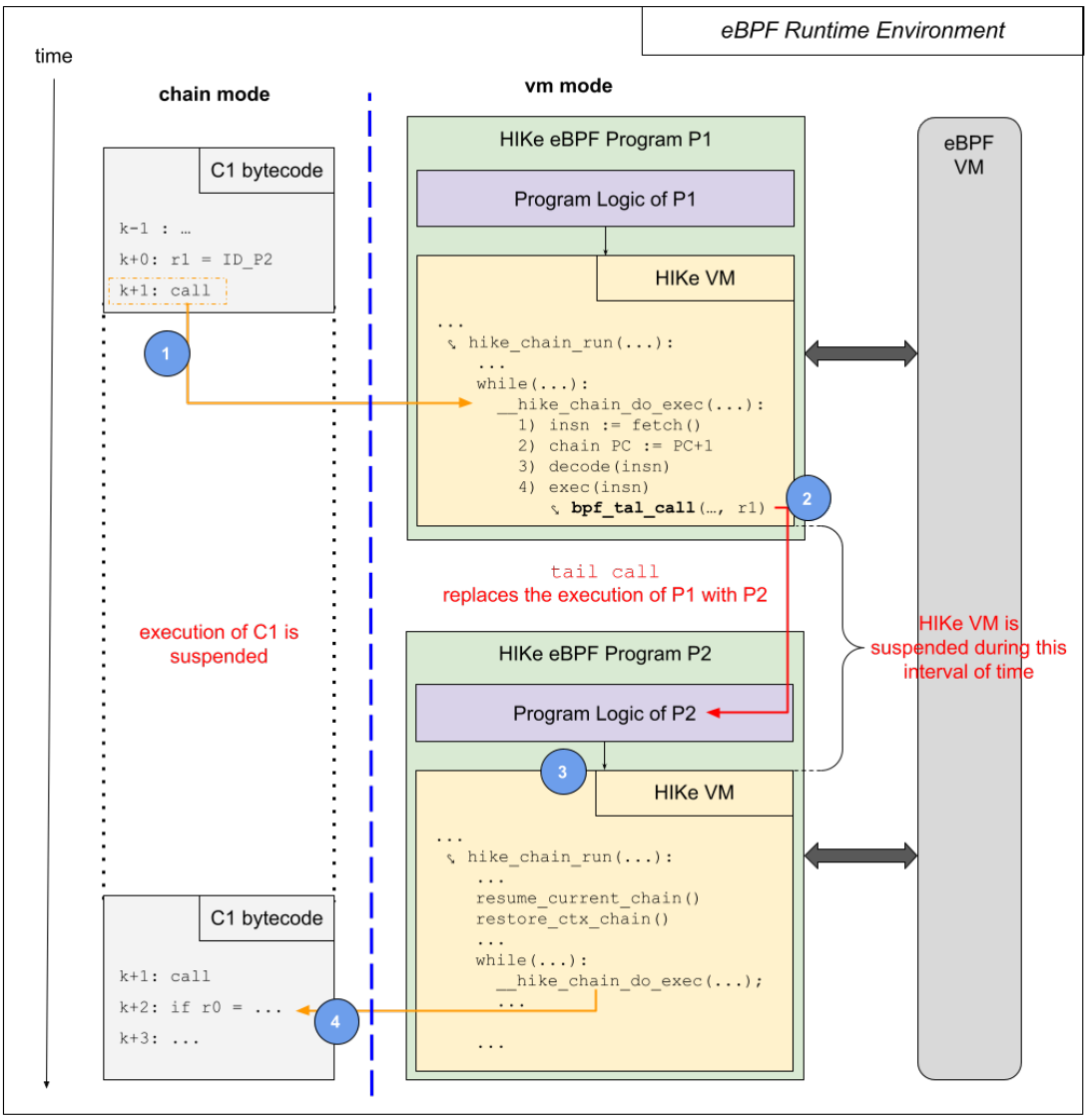


Figure 7.7: *An illustrated view of the execution mode changing as a HIKe Chain requests to run a privileged instruction (i.e. function call to a HIKe eBPF Program).*

eBPF Programs to share some information. SMA is implemented through an eBPF map and is part of the HIKe Persistence Layer. For performance reasons, the SMA is per-CPU only: it means that in a system where multiple CPUs (or logical cores) are available, a HIKe Chain running on CPU k can not use the SMA to share some data with another HIKe Chain running on a CPU j, where $k \neq i$.

The code of a HIKe Chain can access SMA or packet data through different Virtual Memory Addresses (VMAs). The HIKe VM implements a very simple Memory Management Unit (MMU) which receives a VMA and translates it into a *meaningful* address for the eBPF VM. In other words, the MMU transparently remaps the VMA to the actual address where the data to be accessed is present.

Each VMA is 32-bit[8] long and consists of two parts: the first most significant byte

[8] A VMA is 64-bit wide, but currently, only the lower 32-bit are used. For this reason, I consider each VMA to be 32-bit long.

(MSB) of the memory address represents the "segment" or "bank", while the remaining bits represent the offset within that segment. As a result, there can be a maximum of 256 different segments, and in each segment up to 16MB can be addressed.

From a HIKe Chain point of view, every memory address is a VMA and the bytecode (i.e. instructions) contains only reference to VMAs. The HIKe VM uses different "segments" for providing the access to different types of memory. For instance, the $VMA = 0x3000000$ represents the base virtual address of the SMA. Every data read-/written from/to this address involves an access to the eBPF map (i.e. HIKe Persistence Layer) which stores the SMA data. Conversely, the $VMA = 0x1000000$ provides the access from a HIKe Chain to the network packet that is being processed. As can be noted, the two "segments" are different. I designed and implemented such a unified memory model to simplify data access from HIKe Chains, so that Muggles can not go crazy using different techniques to access different types of memory.

Simple and Efficient Packet Handling

Considering the XDP hook, the eBPF VM can read/write the content of the packet (i.e. XDP fame) directly using pointers to the *start* and the *end* of that packet, available through the XDP context (`struct xdp_md`). The HIKe VM reserves two Virtual Memory Addresses (VMAs) that are used for accessing the packet being processed. Such addresses are translated by the Memory Management Unit (MMU) of the HIKe VM into memory addresses pointing to the real data (i.e. the packet data).

For the developer (Muggle) of a HIKe Chain, the final result is that the access to the packet bytes is similar to what is possible to do in an eBPF/XDP program. For example, the `Ethernet type` field of an Ethernet frame can be accessed by a HIKe Chain using the code shown in Listing 7.3. `HIKE_MEM_PACKET_ADDR_DATA` is the VMA of the start of the packet, `nthos()` is a macro that converts the endiness from network byte order to host byte order.

Listing 7.3: *Access to packet fields in a HIKe Chain.*

```
1    __u8 *p = HIKE_MEM_PACKET_ADDR_DATA;
2    __u16 eth_type = nthos(*(__be16*)(p+12));
```

With reference to the Listing 7.3, in a HIKe Chain it is not strictly necessary to verify that a pointer `p + off` refers to a valid memory area as it would be mandatory to do in any eBPF program, under penalty of rejection by the verifier. The HIKe VM automatically checks that any access to any memory location (i.e. VMA) is valid and safe. If it is not, the HIKe VM immediately terminates the execution of the HIKe Chain by taking a default action that causes the packet to be dropped.

However, in packet processing operations it is very common to check for the length of the packet in order to decide whether or not to access specific parts of the packet. For this reason, the HIKe VM makes available to a Muggle the current length of the packet to which the chain is associated. The packet length is mapped to a specific VMA and it is represented by the `HIKE_MEM_PACKET_ADDR_LEN` macro.

Listing 7.4 shows how to access the `Ethernet type` of a packet in a HIKe Chain (offset $+12$ from the beginning of the frame) only if that packet is at least 14 bytes long.

Listing 7.4: *Getting the packet length in a HIKe Chain.*

```
1
2  [...]
3
4  __u32 *pkt_len = HIKE_MEM_PACKET_ADDR_LEN;
5  if (*pkt_len >= 14) {
6      eth_type = [...];
7
8      [...]
9  }
10
11 [...]
```

The HIKe Framework provides some helper functions to wrap the logic shown in Listing 7.4. For instance, the `hike_pkt_read_u16(u16 *p, int off)` returns 0 (i.e. success) if the helper can read 2 bytes from the packet at the offset `off`. The offset is considered with respect to the beginning of the packet. If the returned value differs from zero, it means that the operation failed because the HIKe Chain attempted to read an invalid portion of the packet.

HIKe Chain-to-Program call

The HIKe VM supports the *function call* pattern for invoking a HIKe eBPF Program. Such functionality is implemented in the HIKe VM through several helper functions, each one considering a different number of input parameters as shown in Listing 7.5.

Listing 7.5: *HIKe helper function for calling a HIKe eBPF Program/Chain from a HIKe Chain.*

```
1 u64 hike_elem_call_1(u64 id);
2 u64 hike_elem_call_2(u64 id, u64 arg2);
3 u64 hike_elem_call_3(u64 id, u64 arg2, u64 arg3);
4 u64 hike_elem_call_4(u64 id, u64 arg2, u64 arg3, u64 arg4);
5 u64 hike_elem_call_5(u64 id, u64 arg2, u64 arg3, u64 arg4, u64 arg5);
```

The first input parameter `id` is the Program ID which refers to the HIKe eBPF Program to be called, while the remaining input ones correspond to the arguments expected by that program. As can be observed, the total number of generic input parameters supported is at most 4 (the first parameter always contains the Program ID). The HIKe VM is designed to honor the Application Binary Interface (ABI) of the eBPF core. Consequently, the input parameters to be provided to the *callee* HIKe eBPF Program are passed in the HIKe VM registers `r1-r5` as follows: `id` in `r1`, `arg2` in `r2` and so on up to `r5` in `arg5`. The *callee* HIKe eBPF Program can return an output parameter to the *caller* HIKe Chain by setting the value in register `r0`. All the input/output parameters accepts only[9] scalar types which are 64-bit wide, at most.

To simplify the discussion below, I assume the `hike_elem_call_x` is invoked while the HIKe VM is run in the context of the HIKe eBPF Program P1 and the *callee* HIKe eBPF Program P2 is identified by *IDP2* (Program ID).

The *function call* to a HIKe eBPF Program is considered to be a *privileged* instruction since it changes the internal state of the HIKe VM and affects the Runtime Environment of both HIKe and eBPF. In particular, executing one of the functions listed in 7.5 involves a series of operations that are summarized below:

1. the HIKe VM switches the execution mode from *chain* to *vm*;

[9] Actually also pointers to locations in SMA can be provided to HIKe eBPF Programs since a VMA can be encoded in a `u64` type.

2. the `bpf_tail_call()` (eBPF) helper function is invoked providing the following input parameters: i) the XDP context; ii) the eBPF map containing the binding between the Program ID and the eBPF Program (such map is part of the HIKe Persistence Layer); iii) and the Program ID (i.e. *IDP2*);
3. the eBPF execution flow switches from the HIKe eBPF Program P1 (where the HIKe VM was running) to the HIKe eBPF Program P2.

At this point, the eBPF VM is running the program P2. Since P2 is an HIKe eBPF Program, after the Program Logic P2 is completed, the flow control passes to the embedded HIKe VM. The VM retrieves the suspended HIKe Chain, switches the execution mode back to *chain* and it starts executing that chain bytecode. The HIKe Chain can access the return value of the program P2 through the register `r0`.

It is worth noting that a HIKe eBPF Program can access to the registers of the *caller* HIKe Chain. By contract, the Program Logic of an HIke eBPF Program should touch only the `scratch`, the `r0` registers and not the *callee* saved ones.

HIKe Chain-to-Chain call

The HIKe VM allows a HIKe Chain to invoke another HIKe Chain. Thus, the HIKe VM supports the execution of nested HIKe Chains. From the HIKe VM point of view, a HIKe Chain-to-Chain call is handled similarly to a Chain-to-Program one. In particular, the *function call* paradigm applied to a *callee* HIKe Chain is realized by exploiting the same macros defined in Listing 7.5. The HIKe VM back-end can distinguish between a call to a HIKe eBPF Program and a HIKe Chain by the ID supplied to the helper function.

To simplify the discussion below, I assume the `hike_elem_call_x` is invoked while the HIKe VM is run in the context of the HIKe eBPF Program P1, the HIKe Chain C1 is currently executed, and the *callee* HIKe Chain is C2 whose Chan ID is *IDC2*.

The *function call* to a HIKe Chain is considered to be a *privileged* instruction since it changes the internal state of the HIKe VM and affects the Runtime Environment of HIKe. In particular, executing one of the functions listed in 7.5 involves a series of operations that are summarized below:

1. the HIKe VM switches the execution mode from *chain* to *vm*;
2. the execution context of HIKe C1 is saved into the chain frame dedicated to C1 at the top of the HIKe VM Chain Stack;
3. a new frame is pushed on top of the HIKe VM Chain Stack as it will hold the space for storing the execution context of C2, spill/fill registers and local variables;
4. the HIKe Chain C2 is retrieved from the HIKe Persistence Layer;
5. the register `r10` is set for pointing to the stack frame of C2;
6. the register `r1-r5` values of chain C1 are copied into the execution context of chain C2;
7. the *Executor* starts to execute the first instruction of the HIKe Chain C2.

At this point the HIKe Chain C1 is suspended, while the chain C2 is run by the HIKe VM. Once the HIKe Chain C2 completes, the following operations are required to resume the execution of the HIKe Chain C1:

1. the `r0` value of chain C2 is copied into the correspondent register saved into the execution context (stored into the HIKe VM Chain Stack) of chain C1;
2. the frame of chain C2 is popped out from the HIKe VM Chain Stack;
3. the execution context of C1 is resumed;
4. the *Executor* starts to execute the instruction that immediately follows the Chain-to-Chain call in the HIKe Chain C1.

From the developer (Muggle) point of view, it is possible to group complex functionality in HIKe Chains by creating logical functional structures that simplify both the writing and the management of the network application to be implemented. Note that the call to a HIKe Chain does not exploit the *tail calls* mechanism and consequently does not impact on the total limit of available *tail calls* that is set by the eBPF VM to 32. In other words, a number of calls to the HIKe Chain can be made that does not depend on the number imposed by the *tail call* mechanism.

The maximum number of nested HIKe Chains is, however, limited by the depth of the HIKe VM Chain Stack that currently provides for a maximum nesting of 8 HIKe Chains. This value can be changed by the Wizard during the compilation of the HIKe eBPF Programs.

7.6.2 HIKe eBPF Program

HIKe eBPF Programs are the basic components of the HIKe framework. They are regular eBPF Programs coded in C language that are *decorated* (i.e. embedding HIKe VM code) so that they can be integrated in HIKe Chains. Each HIKe eBPF Program is made of two parts: the Program Logic and the HIKe VM code. The Program Logic is the business logic that a developer (Wizard) realizes for processing a packet. Instead, the HIKe VM code aims to execute the bytecode of the HIKe Chain to which the program belongs. Since the HIKe VM code is generic, the HIKe eBPF program can be reused simultaneously in many different HIKe Chains.

A HIKe eBPF Program comes with some peculiarities with respect to *traditional* a eBPF program. Specifically, a HIKe eBPF Program may accept a maximum of 5 input parameters (handed through registers `r1-r5`) which are provided by the calling HIKe Chain and it terminates by returning the execution control to the HIKe VM, specifying a return code and an output parameter (stored in `r0`). On the basis of the return code, the HIKe VM can take different actions, i.e: deliver the packet to the kernel, drop the packet or continue with the execution of the chain. The output parameter can provide the calling HIKe Chain with information that can later affect its execution. For instance, a HIKe eBPF Program can return a value to the calling HIKe Chain used for accepting or dropping the packet that is being processed.

A HIKe eBPF Program (written in C) is compiled in an ELF object that can be injected into the eBPF runtime. In the HIKe Runtime Environment (HIKe RE), each loaded HIKe eBPF Program is bound to a unique Program Identifier (Program ID) and

the binding information (ID, Program) is stored in a special map (array program map) within the HIKe Persistence Layer. Program IDs are used to concatenate the referring programs, as these identifiers can be used in HIKe Chains. The eCLAT framework is capable to handle automatically the compilation, loading and registering of a HIKe eBPF Program in both the eBPF and HIKe REs as discussed in Subsection 7.6.5.

The source code of a HIKe eBPF Program is shown in Listing 7.6. Two macros (i.e. pre-processor directives) are used to transform an eBPF program into a HIKe eBPF Program.

Listing 7.6: *A simple HIKe eBPF Program.*

```
1  HIKE_PROG(allow)
2  {
3      bpf_printk("HIKe allow Program");
4      bpf_printk("Program ID=%llx, Accepted proto=%x",
5                 HVM_ARG1, HVM_ARG_2);
6
7      return XDP_PASS;
8  }
9  EXPORT_HIKE_PROG_2(allow, u16, proto);
```

The `HIKE_PROG` macro (line 1) is used for declaring a HIKe eBPF Program. Such a macro accepts the name of the program that could be whatever string decided by the Wizard. It is worth noting that HIKe eBPF Programs are identified in the HIKe RE through the assigned Program ID and not by the name. Program shown in Listing 7.6 prints the Program ID and the protocol provided by the calling HIKe Chain ID as parameters. Due to the calling conventions followed by HIKe, the first parameter is set into the register `r1`, the second in `r2` and they are accessed using two macros, respectively `HVM_ARG1` and `HVM_ARG2` (line 5). Those registers have been filled with the actual values by the *caller* HIKe Chain. Moreover, the `EXPORT_HIKE_PROG_2` (line 9) is used for informing the HIKe VM about the number of input parameters expected by this HIKe eBPF Program. Specifically, such program receives 2 input parameters: i) the first one is implicit, always present in any HIKe eBPF Program and it represents the Program ID; ii) the second parameter depends on the specific logic of the program and, in this case, it represents the protocol (`u16 proto`).

The HIKe eBPF Program shown in Listing 7.6 is considered to be *final* since it returns the `XDP_PASS` code to the HIKe VM. Such code forces the HIKe VM to terminate the chain execution and passes the control to the kernel networking stack. Instead, a *non-final* program should use the `HIKE_XDP_VM` code to force the HIKe VM continuing in the execution of the chain.

In addition, this considered HIKe eBPF Program does not return any output parameter to the `caller` HIKe Chain. If a HIKe eBPF Program wants to provide an output to the chain, it must set the `r0` register with the value to be returned: the Wizard can make use of the `HVM_RET` macro for assigning `someval` to register `r0`, i.e.: `HVM_RET = someval`.

7.6.3 HIKe Chain

A HIKe Chain is written in C language. The source code (snippet) of a HIKe Chain is shown in Listing 7.7. The code is compiled into a bytecode that is subsequently stored in the HIKe Persistence Layer, as discussed in Subsection 7.6.5. During this registration phase, an unique Chain ID is obtained and associated with the chain itself:

it will be used for referring to that HIKe Chain from the HIKe VM, the HIKe Chain Loader and the other HIKe Chains.

From the point of view of the HIKe VM, a HIKe Chain is an executable bytecode based on the HIKe VM instruction set. To make the life easier for the HIKe developers, the definition of a HIKe Chain in C language is extremely simple and intuitive thanks to the use of special macros that are provided by the HIKe framework.

Listing 7.7: *Definition of a HIKe Chain (C code).*

```
1  #define allow(ETH_TYPE)                          \
2  hike_elem_call_2(HIKE_PROG_ALLOW_ANY,            \
3                   ETH_TYPE)
4
5  #define mon_ipv6_pkt()                           \
6  hike_elem_call_2(HIKE_EBPF_PROG_PCPU_MON,        \
7                   MON_IPV6_PACKET_EVENT)
8
9  HIKE_CHAIN_1(HIKE_CHAIN_MON_IPV6_ALLOW)
10 {
11     [...]
12     /* read Ethernet type from packet */
13     eth_type = ...;
14
15     if (eth_type == 0x86dd)
16         mon_ipv6_ptk();
17
18     /* call the HIKe eBPF Program 'allow'
19      * providing the input parameter
20      * 'eth_type'.
21      */
22     allow(eth_type);
23
24     return 0; /* fallthrough */
25 }
```

The goal of the HIKe Chain, defined in Listing 7.7, is to identify and monitor IPv6 packets and its logic can be easily followed by reading the source code. The chain is labeled `HIKE_CHAIN_MON_IPV6_ALLOW` and it can be either the real Chain ID or a placeholder used for distinguishing among possible other chains defined in the same ".c" file. In case of placeholder, the label will be later fixed with the real Chain ID as soon as the HIKe Chain will be registered in the HIKe Runtime Environment.

Looking at the code reported in Listing 7.7, it first reads the `Ethernet type` from the packet (the details for reading a field from the packet are omitted for the sake of clarity). If the *Ethernet type* is IPv6 (`0x86dd`), then it calls the HIKe eBPF Program `mon_ipv6_pkt()`, which can perform further monitoring operations specific for IPv6 packets. Finally, the HIKe eBPF Program `allow()` is called, which accepts the read `Ethernet type` as a parameter. Note that this program could perform other operations depending on the Ethernet type of the packet, without the burden of accessing again to the field of the packet. The `allow()` program terminates the execution of the HIKe Chain by forcing the HIKe VM to accept the packet and, for this reason, it is said to be *final*.

This very simple example has shown the combination of two already compiled HIKe eBPF Programs by means of the *function call* pattern avoiding the eBPF verifier, which would have been impossible without the HIKe VM and the Runtime Environment. With the same programming ease, a HIKe Chain can invoke other HIKe Chains allowing for multiple nested chains.

As can be observed from Listing 7.7, a HIKe Chain is defined by means of the macro `HIKE_CHAIN_1()`. Thanks to this macro, the developer is relieved from the need to deal with the signature of the function that represents a HIKe Chain and with the retrieval of the input parameters. In this case, `HIKE_CHAIN_1(chainid)` declares a HIKe Chain identified by the Chain ID `chainid` which is the only parameter for this chain. The `chainid` is set by the *caller* HIKe Chain or by the HIKe VM if such a chain is the *root* one (i.e. the first chain to be invoked by the HIKe Chain Loader/Traffic Classifier).

The *function calls* to the `mon_ipv6_ptk()` and `allow()` HIKe eBPF Programs are also coded as macros for sake of clarity. Indeed, they both expand with the following HIKe helper function: `hike_elem_call_2()`. This helper function implements the *function call* pattern which: i) invokes the *callee* HIKe eBPF Program; ii) executes its Program Logic; iii) updates the context of the HIKe VM; iv) handles the return code and continues with the chain flow execution. The *function call* is executed in privileged mode by the HIKe VM as I have already thoroughly explained in Subsection 7.6.1.

The HIKe framework also provides macros for defining chains accepting more than one parameter. For example, the `HIKE_CHAIN_2(chainid, arg2_t, arg2)` macro is used to define a HIKe Chain identified by `chainid` that accepts a second parameter of `arg2_t` type whose formal parameter name is `arg2`.

Finally, a HIKe chain is limited in the maximum number of instructions (at the bytecode level), which is currently set to 64.

7.6.4 HIKe Chain Loader

Upon the arrival of a packet on a network interface (configured for HIKe), an eBPF program called HIKe Chain Loader is invoked. The HIKe Chain Loader makes use of the APIs provided by the HIKe VM to associate a HIKe Chain to the packet and start the execution of the bytecode of that chain.

The HIKe Chain Loader selects and executes the correct HIKe Chain based on an arbitrary logic that can be programmed in the loader itself. For example, a HIKe Chain Loader may invoke a given HIKe Chain based on the destination IP address of the packet, or the TOS value and so on. As soon as the HIKe Chain Loader identifies the Chain ID of the HIKe Chain, it calls the `hike_chain_bootstrap()` function, provided by the HIKe framework, passing the Chain ID. Then, the HIKe VM associates the HIKe Chain to the packet and starts executing the bytecode instructions of the chain (which are retrieved from the HIKe Persistence Layer).

7.6.5 HIKe Persistence Layer

The HIKe Persistence Layer (HIKe PL) is a component of the HIKe Runtime Environment (HIKe RE) and aims to solve two fundamental problems: i) store the references to the HIKe eBPF Programs so that the HIKe Chains can call them by their ID (Program ID); ii) identify the HIKe Chains and store their bytecode, so that the HIKe VM can retrieve the bytecode and the HIKe Chains can call other HIKe Chains by their ID (Chain ID).

Under the hood, the HIKe PL consists of several eBPF maps which are *pinned* to the file system so they can *persist* even though no eBPF objects (i.e. eBPF Programs, eBPF

maps) refer to them. Hereafter, I briefly describe the purpose of the maps comprising the HIKe PL:

- **HIKe Program Table**: keeps track of the references between the Program IDs and the HIKe eBPF Programs which are currently available to the HIKe RE;
- **HIKe Chain Table**: contains the bytecode of HIKe Chains which are indexed by their ID (Chain ID). This map catalogs all the chains that are currently available to the HIKe RE;
- **HIKe VM Chain Stack**: is used by the HIKe VM to support the execution of HIKe Chains. This map reflects the structure of a stack. Each stack element (chain frame) contains the execution context of a HIKe Chain that is running (or ready to run) and those that have been suspended. The execution context of a HIKe Chain includes the VM registers, filled/spilled registers, and local variables. The HIKe VM Chain Stack enables, therefore, the support of nested HIKe Chains calls from other HIKe Chains. The maximum depth of the stack is set in the HIKe RE to a maximum of 8 nested chains. However, this value can be changed by a Wizard to support more nested chain calls;
- **HIKe Shared Memory Table**: holds the Shared Memory Area (SMA) used by HIKe Chains, eBPF Programs and HIKe eBPF Programs for information exchange purposes.

The registration of a HIKe eBPF Program or the storage of a HIKe Chain requires the assignment of an ID to the Program or Chain. This is managed by an entity which is external to the HIKe PL and implemented in user space. The main motivation for this design choice is that the operations on the eBPF maps need to be synchronized to avoid concurrent writes leading to inconsistency. It is very hard to achieve synchronization at eBPF level, as there is no way to lock multiple maps at once. It is much cleaner and easier to implement the ID management and the registration operation in a user space program/daemon.

7.6.6 HIKe development process

With reference to Figure 7.8, the required steps to set up a working HIKe eBPF system are summarized as follows:

- **Step 0**: initialization of the HIKe Persistence Layer where all the required eBPF maps are pinned on the file system;
- **Step 1**: the first step is devoted to the loading of the HIKe eBPF Programs. Internally this requires the compilation of the eBPF bytecode, the generation of a Program ID, the loading of the program, the pinning of the program and its maps, and the registration to the HIKe RE (i.e. storing the binding between Program ID and eBPF HIKe Program into the HIKe PL);
- **Step 2**: once all HIKe eBPF Programs have been successfully registered, it is necessary to take care of the HIKe Chains. This requires the compilation of the chains with the IDs of the programs within, the generation of the Chains IDs and their loading in the HIKe Chain Table;

- **Step 3**: the final step requires the compilation of the eBPF Chain Loader (usually a packet classifier). The task of this program is to call the right HIKe Chain according to the classification result and, thus, it needs to be configured with the related Chain IDs. Such IDs can be included directly in the HIKe Chain Loader source code ("hard-coded") or read from a map. Finally, the program needs to be loaded and eventually attached to the XDP hook.

To simplify the development process, several configuration scripts are provided with the HIKe Framework source. However, the eCLAT framework takes care of all the steps necessary to properly deploy and initialize the HIKe RE.

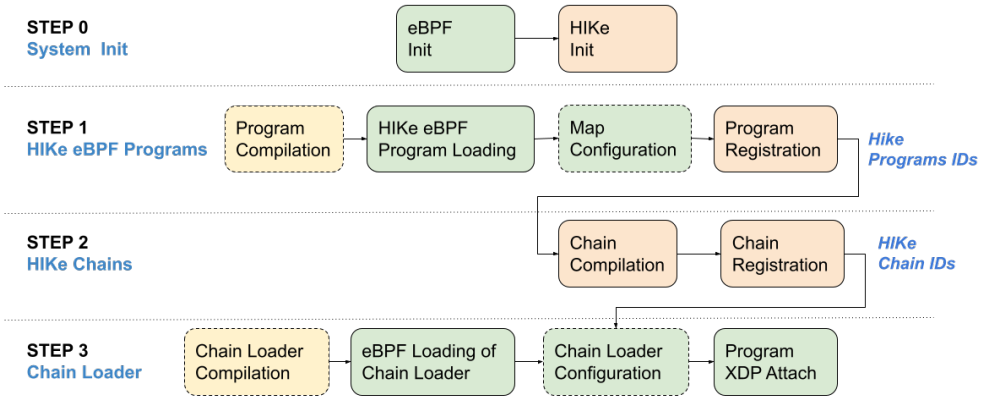


Figure 7.8: *Steps required to set up a complete eBPF/HIKe system*

7.6.7 Code portability

HIKe Chains are written to run independently from the kernel versions. While in principle HIKe eBPF Programs may suffer from portability problems due to different kernel internals chaining from version to version, in practice this is not the case since the API for eBPF/XDP is quite stable. However, recently eBPF designers have been trying to improve the portability of eBPF applications by introducing the concept of "Compile Once - Run Everywhere" (CO-RE) [45]. It aims to keep eBPF programs compatible, as much as possible, with different versions of the Linux kernel without necessarily requiring changes to the source codes.

On the contrary HIKe Chains do not have any portability problem because the HIKe VM takes care of running the HIKe Chains adapting to the kernel version, to the available libraries etc. With HIKe Chains, the concept of "write once, run everywhere" is pushed even further.

7.7 The eCLAT abstraction

HIKe introduces programmability and modularity in eBPF, however the technical entry barrier for a developer that wants to implement a custom logic in eBPF network functions is still high. For this reason the eBPF Chains Language And Toolset (eCLAT) framework was designed and implemented. eCLAT is a framework and a language that

supports eBPF/HIKe based network programmability offering a high level programming abstraction.

eCLAT simplifies the reuse of the HIKe eBPF Programs by defining a python-like scripting language used to specify all HIKe operations, such as composing (HIKe) eBPF Programs and configuring eBPF maps, hiding the difficult technical details, i.e. avoiding the need of low level programming eBPF using the C language and simplifying the retrieval, compilation and injection of HIKe eBPF Programs.

In a nutshell, eCLAT provides an easy-to-use python-like language to simplify the writing of HIKe Chains, avoiding the need of explicit interaction with the HIKe Persistence Layer.

7.7.1 Features

I summarize the key features of the eCLAT framework as follows:

- **Easy-to-use language to express HIKe Chains** using a python-like scripting language, and providing a library for easy access to the packet fields;
- **Download and manage the HIKe eBPF Programs** by automatically collecting the imported programs from the eCLAT code repository, taking care of compilation and bytecode loading. Calling another HIKe eBPF Program/HIKe Chain appears to the eCLAT developer as a conventional Python function call, masquerading the needs to register and use numerical Program IDs/Chain IDs needed by the HIKe Persistence Layer;
- **Simple access to BPF maps**, viewed as simple python dictionaries;
- **Define an entry point** by specifying the HIKe Chain Loader/Traffic Classifier responsible to trigger the HIKe VM, and its configuration. As an example, a classifier based on the IPv6 destination address was provided: according to the IPv6 destination address, different HIKe Chains can be executed.

7.7.2 Architecture

Figure 7.2 shows the architectural view of eCLAT. eCLAT has been implemented in Python as a daemon (*eclatd*). The eCLAT daemon receives user commands from a CLI (`eclat`) through a gRPC interface. The structure of the data is described through a protocol buffer language [41]. Through the CLI, Muggles can load an eCLAT script which instructs the daemon to i) import all necessary HIKe eBPF Programs by collecting their code, compile, inject and register them to the HIKe Runtime Environment (HIKe RE); ii) translate the high-level code of the chain in C language, compile and load them in the HIKe RE; iii) manage the entry point (HIKe Chain Loader) by retrieving its code, compile, attach to XDP hook, and configure according to custom parameters. The eCLAT CLI allows Muggles to query the daemon about the current status of eBPF maps. The daemon based architecture is also consistent with a security limitation of eBPF runtime which grants only to the process that mounted the eBPF file system (or one of its children) to have access to the filesystem itself. In this case, the eCLAT daemon initially mounts the eBPF filesystem and then is responsible for all related operations.

As shown in Figure 7.2, the eCLAT daemon is composed by the following functional blocks:

- **Protocol engine**: implements the gRPC protocol service and is responsible for the communication with the CLI;
- **Controller**: is responsible to set up the networking environment, to interact with the parser and to execute the scripts invoking the managers. It generates/retrieves IDs for HIKe eBPF Programs and HIKe Chains. Such identification numbers will be fundamental for the chain compilation phase since the HIKe Chains rely on numerical IDs for calling HIKe eBPF Programs, rather than on the names which are used in the eCLAT domain;
- **Program**: wraps and manages a HIKe eBPF Program. The component fetches programs from the eCLAT public repository, compiles them, and takes care of the loading and unloading operations. Finally, it registers the output in the HIKe Persistence Layer. It means that the associations between Program ID and the specific HIKe eBPF Program are kept safe in the HIKe Persistence Layer to be accessed, later, by both HIKe Runtime and Virtual Machine. During the compilation of the HIKe eBPF Programs, the debug info about the program (i.e.: variables, functions, structs, etc.) are automatically extracted and registered in a JSON file. This file is parsed to obtain all map/program associations as well as the number of input parameters accepted by the specific HIKe eBPF Program;
- **Chain**: handles the script part related to HIKe Chains. It is in charge of translating the source code, from the (python-like) eCLAT script to a C-defined HIKe Chain. Then compiles it to generate artifacts (i.e. ELF file object) through the execution of a dedicated Makefile. Finally, it registers the output in the HIKe Persistence Layer. The HIKe Persistence Layer contains a catalog between all the HIKe Chains loaded (and thus their bytecodes) and the Chain IDs assigned by the eCLAT Runtime Environment;
- **Chain Loader**: this component handles one or more HIKe Chain Loader(s) and interacts with their maps. Using the eCLAT scripting language, a Muggle can specify the chain loader that has to be loaded, attached to the XDP hook as well as the configuration that has to be enforced through configuration maps;
- **Parser**: has the task of analyzing the eCLAT scripts and creating the Abstract Syntax Tree (AST), in order to interpret the provided commands and generate the C code which defines the HIKe Chains;
- **Command Abstraction Layer**: provides an abstraction over the different shell commands that need to be invoked on the operating system to deal with the eBPF and HIKe REs.

7.8 Evaluation

7.8.1 Prototype

To test both HIKe and eCLAT frameworks, a fully functional prototype has been implemented and delivered under the form of a single Docker container, available at [68].

Within the container, it is possible to develop and test HIKe eBPF Programs and eCLAT Chains. In particular, the prototype aims to emulate a node implementing the HIKe/eCLAT framework and a node that generates traffic to be processed. The Docker container is pre-configured for supporting the experiment with the DDOS example coded in Listing 7.2. Several HIKe eBPF Programs are also provided to demonstrate how they can be easily combined in eCLAT Chains to implement fairly complex packet processing scenarios. The source code and the reference documentation are respectively available at [160], [69], and provided as supplementary material.

7.8.2 Modularity

Table 7.1: *Comparison of the modularity features for different eBPF frameworks.*

Dimension	**Cilium**	**Polycube**	**eCLAT**
Define application logic	configuration and API	configuration of modules and topology	programmatic
Composition approach	assembling and compiling different building blocks	interconnection of cubes through ports (e.g., veth pairs)	dynamic composition of eBPF programs with no re-compilation.
Composition topology	-	linear (tail call)	arbitrary
Code generation	BCC-based	BCC-based	transpiled from eCLAT script to C, and compiled with CLang/LLVM
Modularity	pre-defined programs	big modules (cubes)	any eBPF program
Extensibility	submit a patch to the main project	creation of a new cube within the framework	conventional eBPF programs with minor modifications

The greatest benefit of adopting the HIKe/eCLAT framework is in the flexibility and modularity it offers. Table 7.1 objectivize the benefits of this approach by comparing the presented solution with popular frameworks, Cilium and Polycube, across different dimensions, and specifically:

- **Define application logic**: how a user (Muggle) can define/implement a custom application logic? eCLAT allows users to define their business logic in a programmable way through eCLAT scripts. Conversely, other frameworks allow defining custom configuration. The difference is that programming flows allow much more expressibility than relying on a pre-defined set of parameters to configure;
- **Composition topology**: which topology of the data pipeline is supported by the framework? eCLAT supports arbitrary topology as the data flow can follow different branches and loops. Other frameworks like Polycube are limited by a linear topology: data passes through a predefined set of eBPF Programs which hands over through a set of *tail calls*. If developers want to implement a custom calling logic on the basis of specific interaction patterns (i.e. a program calls another program accordingly to given conditions), they must do it on their own;
- **Composition approach**: where the composition of different modules happens? eCLAT is the only one which permits composition *inside* eBPF, without requiring

any eBPF Programs (modules) recompilation. This is different from models where the composition happens in user space and then, through code generation, eBPF programs bytecode is injected;

- **Code generation**: differently from others, eCLAT is *not* based on BCC [13] but CORE [45] which is fostered and maintained by the Linux kernel community;

- **Modularity**: Which is a module? for Cilium there are pre-defined generated programs, and Polycube relies mainly on "big" modules (i.e. "the firewall") as they can be only chained together. Conversely, for eCLAT a module is a standalone HIKe eBPF Program that can be also quite small (i.e. "flow meter") as its utility must not be absolute, but functional of the context where it will be placed in the HIKe Chain (i.e. in an *if* expression to decide a branch);

- **Extensibility**: How can a user (Wizard) create a new module to extend the framework? HIKe/eCLAT module can be any legacy eBPF program with very few changes (3 or 4 lines of C needs to be added). Extending other frameworks requires more skills.

7.8.3 HIKe data plane performance evaluation

Which is the performance penalty in using HIKe abstraction instead of low-level coding in eBPF? To answer this question, I focus on data plane performance and I consider the HIKe framework and the HIKe VM. I measured the processing overhead caused by executing (interpreting) operations inside the HIKe VM, which in turn is executed by the eBPF VM. I expect that writing the Chains using eCLAT will not decrease performance, because the transpiling operation from eCLAT to C does not introduce overhead and the transpiled C code is compiled as a HIKe Chain.

My performance evaluation consists in measuring the maximum forwarding throughput of a node. This is defined as the maximum packet rate (measured in packets per second - pps) for which the Packet Drop Ratio (PDR) is smaller than or equal to 0.5%. For further details on the methodology adopted to carry out these experiments, see Chapter 3.

Testing scenario: DDoS mitigation with HIKe

I consider a realistic DDoS mitigation scenario which is a popular application of eBPF on XDP [15]. I assume that a software Linux based router is forwarding packets and needs to identify the packets that belong to a *blacklist* of source IP addresses. In particular, packets in the blacklist have to be marked with a given IP TOS. The marking program adds a processing burden to the normal forwarding operations, the obvious goal is to keep this burden as low as possible. Hereafter, I developed and compared three solutions for the marking program: i) a conventional eBPF program (raw); ii) a HIKe eBPF Program (hike); and iii) an IP set [73] based approach (ipset). Source codes for the different eBPF solutions are provided at [160]. The eBPF raw solution is the most difficult to be programmed, only the Wizards can do it. The HIKe solution could even be developed by the Muggles using eCLAT. The IP Set solution has an intermediate development complexity.

	Raw eBPF	HIKe	IP set
Av. Throughput (kpps)	**2570**	**1867**	**991**
Std. Dev. (kpps)	2.1	3.9	1.4
overhead w.r.t. raw eBPF	-	27.7%	61.4%
overhead w.r.t. HIKe	-	-	46.7%

Table 7.2: *Performance of DDoS implementations.*

Table 7.2 reports the achieved speed in terms of maximum throughput. As can be observed, HIKe presents a $\sim 27.7\%$ of performance degradation with respect to the raw eBPF approach (1.8 mpps versus 2.6 mpps). However, HIKe outperforms IP set whose performance is $\sim 46.7\%$ worse than HIKe and $\sim 61.4\%$ worse than the raw eBPF solution. The results are basically affected by the cost of two components: i) the operations to support the infrastructure (i.e. eBPF, HIKe, Netfilter/Xtables) used to run the network programs/applications; ii) the operations implemented by the business logic of application that depend on how traffic needs to be processed. The cost of (i) is independent of the business logic of the network applications and, thus, can be considered as a constant that is always present, in *background*. The more expensive the operations and logic implemented by the traffic processing infrastructure, the greater is the negative impact on overall performance. In parallel, however, the cost of (ii) depends on the specific application. If a network application is simple and does not require expensive operations such as changing significant portions of the packet, then the burden of the infrastructure used to run that application may become significant. Conversely, if the application requires operations such as traffic encapsulation or header inspection, the cost of such operations may dominate that imposed by the processing infrastructure. Chaining of programs in eBPF is accomplished by writing ad-hoc code in each program considering the specific case. In HIKe, on the other hand, concatenation of HIKe eBPF Programs is achieved through VM code that interprets the bytecode of a HIKe Chain (i.e., arbitrary concatenation logic). The HIKe VM code is, therefore, much more complex than ad-hoc code and requires many more machine instructions (at eBPF level) in order to be implemented. Consequently, the greater the cost of Program Logic implemented in HIKe eBPF and raw eBPF programs, the narrower the performance gap between HIKe and eBPF.

Testing scenario: HIKe worst case overhead

Different use cases and programs to be composed lead to different estimations of the overhead introduced by HIKe compared to the raw eBPF based solution. In this regard, I consider a synthetic reference scenario that represents the worst case situation for HIKe. In fact, I compose a number of *dummy* programs that "do nothing", exploiting *function calls* to HIKe eBPF Programs in HIKe Chains, and *tail calls* in (raw) eBPF programs. To this aim, I setup a simple "for cycle" benchmark program implemented with *tail calls*. The eBPF program reads an integer value i on a eBPF map and decrements it; then the program exits if $i = 0$ or *tail call*s itself if $i > 0$. Conversely, a HIKe Chain implements the "for cycle" and is in charge of calling the dummy HIKe eBPF Program.

Figure 7.9 shows the max throughput of both HIKe VM (hike) and the eBPF pro-

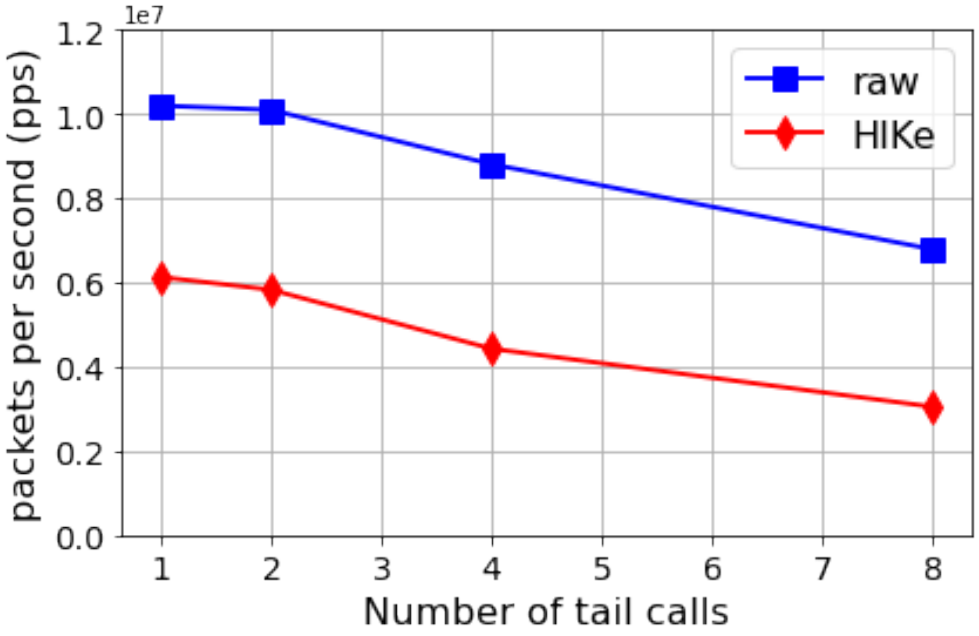


Figure 7.9: *HIKe vs. raw eBPF performance considering the worst-case scenario.*

grams (raw), which decreases with the number of *tail calls* performed. The HIKe VM presents a relative performance degradation of $\sim 40\%$ in case of 2 *tail calls* (10 mpps vs 6.1 mpps), and reaches $\sim 54\%$ when 8 *tail calls* have been considered (6.8 mpps vs 3.1 mpps). Roughly, I can estimate that the cost of composing such programs under the HIKe VM is twice the cost of raw eBPF *tail calls*. This estimation corresponds to the worst case scenario where programs implementing very minimalist logic processing logic are chained together. Hence, considering the practically absent application logic of these programs, this test provides an estimation of the infrastructure overhead of HIKe compared to eBPF. I am very satisfied with this result, considering that: i) the HIKe VM is an interpreter while the eBPF code is JITted on the X86-64 arch; ii) I have a substantial gain in programming easiness; iii) I am already working to improve the performance with respect to my first prototype; iv) the current performance is already adequate in several real scenario scenarios as shown in the previous subsection.

7.9 Related Work

eBPF has been extensively used for building fast and complex applications in several domains such as tactile [183], security [180], cloud computing [88] and network function virtualization [116]. In what follows, I confined the analysis on the limitations of the system and on the relevant framework.

7.9.1 eBPF limitations and investigations

eBPF provides advantages to network programmers but it also presents several limitations that have been highlighted by researchers and often tackled to provide mitigation or propose re-design. A comprehensive review of eBPF technology opportunities and

shortcomings for network applications is provided in Miano et al. [111] that analyzes the use of eBPF to create complex services. The authors pinpoint the main technological limitations for specific use cases, such as broadcasting, ARP requests, interaction between control plane and data plane, and when possible they identify alternative solutions and strategies. Some of the problems reported in [111] are part of the motivations that led to the design and development of HIKe and eCLAT. Gershuni et al. [60] analyze a design of eBPF in-kernel verifier with a static analyzer for eBPF within an abstract interpretation framework, to overcome the current verifier limitations. The authors' goal is to find the most efficient abstraction that is precise enough for eBPF Programs. Their choice of abstraction is based on the common patterns found in many eBPF Programs with several experiments that were performed with different types of abstractions. I also recognize the relevant role of the "validation hell" and I believe that the HIKe architecture can help to mitigate the problem.

7.9.2 eBPF frameworks for networking

There are several eBPF based projects and frameworks devoted to simplify or manage the networking using eBPF. The most popular ones are three: Polycube, Cilium and Inkev.

First of all, HIKe is an evolution of the HIKe-v0 that I presented in Chapter 6. Indeed, v0 lays the foundation of eBPF program chaining that is achieved specifying a list of ID (Program ID) to be called. Specifically, the chain topology is linear: programs are invoked strictly one after another. Moreover, the programmer must manually insert into the code of each eBPF Program the provided macro which reads the next program ID from the chain and *tail call*s it. This is all because there is neither a Virtual Machine nor the concept of *function calls* in the v0 version of HIKe.

Polycube aims to provide a framework for network function developers to bring the power and innovation promised by Network Function Virtualization (NFV) to the world of in-kernel processing, thanks to the usage of eBPF [109, 110]. Network functions in Polycube are called Cubes and can be dynamically generated and inserted into the kernel networking stack. Like in HIKe/eCLAT, Polycube is devoted to implement complex systems through the composition of cubes. However, Polycube's goal is not to reconstruct functional programming but to build chains of independent micro-services. The absence of function calls does not allow eBPF Programs to return values or accept input arguments, and thus it is not possible to change the flow logic according to the output of a given program.

Another approach for using eBPF inside the NFV world is provided by Zaafar et al. with their InKeV framework [7]. *InKeV* is a network virtualization platform based on eBPF, devoted to foster programmability and configuration of virtualized networks through the creation of a graph of network functions inside the kernel. The graph which represents the logic flow, is loaded inside a global map. The logic implemented by the graph is merely related to the function composition, while I provide a more complex flow within the HIKe VM (e.g., branch instructions, loops, and in general programmable logic). Such as for Polycube, the goal of InKeV is to provide network-wide in-kernel NFV, which is not the HIKe framework main goal but that can certainly be one of the most important applications of it.

Cilium is an open source application of the eBPF technology for transparently se-

curing the network connectivity between cloud-native services deployed using Linux container management platforms like Docker and Kubernetes [129]. With respect to this work, Cilium has a totally different target as it is focused on the security of applications running in containers. Conversely, my target is the reusability of different eBPF Programs and their composability inside the chains, separating the composition logic flow from the HIKe eBPF Programs themselves. I think big applications like Cilium could greatly benefit from this new approach.

Risso et al. proposed an eBPF-based clone of Iptables [16]. The approach uses an optimized filtering based on Bit Vector Linear Search algorithm which is a reasonably fast and consolidated programming interface based on Iptables rules. Clearly, the focus of the work is not composability, but an extended version of such an approach could be used to define the entry point for the HIKe applications.

It is worth mentioning the application of eBPF to provide a greater flexibility to Open vSwitch (OVS) Datapath [164, 165]. The works propose to move the existing flow processing features in OVS kernel datapath into an eBPF program attached to the TC hook. They also investigate how to adopt kernel bypass using `AF_XDP` and moving flow processing into the user space. HIKe/eCLAT can be used inside OVS as well but it is not my focus to specialize the framework for a particular application. Finally, several authors implement eBPF Hardware Offload to SmartNICs [24, 80].

7.10 Conclusions

Using HIKe, the eBPF *Wizards* can provide eBPF Programs that can be easily composed by *Muggles*. The Muggles write high level programs (Chains) in the eCLAT scripting language and framework.

The HIKe VM executes the Chains without passing through the eBPF verifier. Programmers can thus call eBPF programs as "simple" *function calls*. The benefits of the HIKe architecture are twofold: i) a HIKe eBPF Program can be reused "as is" in several different application contexts with no code change needed and ii) different application logic can be implemented just by writing HIKe Chains as the HIKe VM takes care of the runtime composition.

eCLAT helps network programmers in the creation of complete applications that mesh up the programs made by wizards. Both HIKe and eCLAT frameworks are available under a liberal open source license at [160].

The overhead of the HIKe architecture has been evaluated in terms of computational overhead and performance degradation considering a typical scenario and a worst case. The HIKe performance is already better with respect to competing tools for Muggles (i.e. Netfilter base solution such as IP set). The performance penalty with respect to raw eBPF programming is already acceptable considering the huge gain in simplicity for some applications. I expect that further work on the HIKe framework will also reduce its gap with respect to raw eBPF.

Conclusions

The telecommunications world is evolving at a dramatic pace. The need to satisfy ever-increasing requirements such as demand for higher bandwidth, resilience to failures or attacks and support for the unstoppable growth of connected devices, places telecommunications networks at the center of attention of both industry and academia. Software Defined Networking (SDN), coupled with Network Function Virtualization (NFV) and Cloud-Edge-Fog Computing are the key paradigms for today's networks such as 5G and the ones coming in the near future. These technologies provide a full "softwarization" of the network that enables an unprecedented level of programmability and flexibility of devices, applications and services. Nevertheless, core routers and network equipment located at data centers still maintain a large amount of information and state necessary to support delivered services, affecting scalability.

Segment Routing on IPv6 elegantly solves most of the problems related to maintaining the state of information within the network, and at the same time, provides incomparable network programming capabilities. The advantage of SRv6 is the ability to add state information in packet headers, avoiding or minimizing the information that is configured in internal nodes for implementing network services. This information is encoded in instructions or segments (IPv6 addresses) that are associated with functions to be executed as the packet crosses the network. Such instructions form network programs that are added to packets at the edge, without involving the core network.

My research work has started by studying, extending and improving the network programmability model introduced by SRv6 within software routers based on the Linux kernel. In current networks, Linux-based routers are located both at the edge of networks as they are used extensively in Fog-Edge Computing, and inside data centers. Through so-called white-boxes, it is possible to realize packet processing systems that combine high performance of dedicated network cards/chips with the networking flexibility and capabilities of the Linux kernel for implementing complex network functions.

In Chapter 2, I studied how the Linux kernel processes network traffic by using different subsystems. In particular, I focused on the SRv6 subsystem, which only partially implements the Network Programming Model of SRv6. I analyzed its functionality and limitations. As a result, I studied and implemented a number of structural changes to

the Linux kernel networking stack that allowed me to implement new networking functions, which are called behaviors in SRv6 jargon. Specifically, I implemented several behaviors to realize L3 VPN services for both IPv4 and IPv6 that use Virtual Routing and Forwarding (VRF) in Linux. This feature is of great importance both for network operators that can offer advanced VPN services, and for data centers that can interconnect different applications of the same tenant leveraging the same layer-3 (IPv6) technology. In addition, I designed and implemented an efficient system of counters associated with each behavior to obtain statistics (packets processed, discarded, etc.) needed for implementing monitoring and performance applications. Network programs with many segments (SIDs) require considerable overhead within the Segment Routing Header, for each packet. For example, 10 segments requires 160 bytes. Therefore, extensions to the network programming model of SRv6 have been proposed with the objective of introducing segment compression techniques to save space in packet headers. One compression technique is Micro SID which, however, was not yet supported by the Linux kernel. Given both the theoretical and practical relevance of this technique, I designed and implemented the Micro SID processing within the Linux kernel networking stack. All the solutions that I proposed in Chapter 2 have become patches merged into the Linux kernel mainline, given their value. The only exception is for Micro SID which is still under review, at the time of this writing.

SRv6 is considered one of the enabling technologies for both operator networks and data centers, and for this reason is crucial to investigate non-functional characteristics such as performance and scalability. Hence, in Chapter 3, I described the design and implementation of SRPerf, a performance evaluation framework for SRv6 implementations. In SRPerf, I provided the evaluation methodology and implemented the algorithms required to estimate performance metrics such as Non Drop Rate (NDR) and Partial Drop Rate (PDR). Using the SRPerf framework, I was able to validate and test the performance related to the new SRv6 Endpoint behaviors and SRv6 capabilities (i.e. counters) that I introduced in the Linux kernel mainline. I also used SRPerf to successfully test the SRv6 uN behavior coming with the new Micro SID extension to the Network Programming Model.

In Chapter 4, I exploited one of the main features of SRv6 consisting in the native support of NFV/SFC scenarios. The Network Programming Model of SRv6 offers the possibility of virtualizing any service by combining the basic functions in a network program that is embedded in the packet header. In this regard, I designed and implemented a *dynamic* proxy mechanism to support Service Function Chaining based on SRv6 for legacy VNFs, a use case of great importance for service providers. My proposed solution was released as open source and extended the current Linux kernel implementation of the SRv6 Network Programming Model. The SRv6 proxy has been smoothly integrated in the Linux ecosystem, for example it can be easily configured through the well known `iproute2` user space utility. I have thoroughly analyzed several performance aspects related to my implementation of the *dynamic* SRv6 proxy. I went through two design and implementation cycles, referred to as SRNKv1 and SRNKv2. I identified a scalability issue in the first design SRNKv1, which has a linear degradation of the performance with the number of VNFs to be supported. The root cause of the problem was the Linux Policy Routing framework. The final design SRNKv2 solved the problem, by extending the Policy Routing framework with the

introduction of a new rule specific for SRv6.

In Chapter 5, I proposed an architecture for supporting Performance Monitoring of SRv6 networks (SRv6-PM). This architecture relies on open source components used for building the cloud native part for data collection and processing, the control plane and data plane parts focused on Loss Measurements in SRv6 networks. With regard to the data plane, I focused on an accurate Per-Flow Packet Loss Measurement (PF-PLM) framework based on the extension of the TWAMP protocol and the alternate marking techniques. I designed and realized several PF-PLM solutions which rely on the Linux kernel networking stack. In particular, I implemented the marking and packet loss counters leveraging different packet processing frameworks inside the kernel. My first implementation of PF-PLM was entirely based on the Netfilter/Xtales/Iptables framework but, because of the way Iptables handles rules, it did not scale well with the number of traffic flows to be monitored. Hence, I came up with a second implementation of PF-PLM that leverages the *IP set* framework still based, under the hood, on Netfilter. In this solution, the overhead introduced by this monitoring solution does not depend on the number of the monitored SRv6 flows. Even though the *IP set* implementation scales well with the number of monitored flows, the overall processing overhead introduced by the monitoring is roughly about 15.0% which is by no means negligible. At this point, I moved to a different design based on a new technology that exploits an in-kernel virtual machine (eBPF VM) able to run network programs. With this approach, packet marking and counting were carried out by ad-hoc eBPF programs. In this case, I was able to achieve a significant performance increase over the *IP set* implementation, getting closer to the basic maximum throughput achieved for a flow that is not monitored. Indeed, the overall overhead introduced by the *eBPF* based PF-PLM solution was less than 5.0%.

In Chapter 6, I proposed an architecture that aims to provide a modular approach to combine different network programs in order to perform complex and customizable packet processing operations. I considered a Linux-based software router in an Hybrid SDN/SRv6 architecture exploiting the eBPF (extended Berkeley Packet Filter). Hence, I designed and implemented a data plane architecture called HIKe-v0 (HybrId Kernel/eBPF forwarding, version 0) which integrates the packet forwarding and processing based on "traditional" Linux kernel networking stack with the ones based on the eBPF VM (i.e. through XDP and TC hooks), taking the best of both worlds. The HIKe-v0 architecture provides an organisation of eBPF programs in the SDN context, overcoming the many technical limitations given by using the XDP fast path and the Linux Traffic Control. HIKe-v0 facilitates the developer in defining lists of eBPF programs to be applied sequentially on packets that match user-defined criteria. Thus, I implemented the architecture in a proof-of-concept, and used 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. This is due to the fact that encap/decap operations for adding/removing SRv6 Policies were implemented as eBPF programs and composed with marking and counting eBPF programs, all in the XDP fast path. On commodity hardware, I obtained a 5x performance increase over the best solution provided in Chapter 5 (5.1 Mpps vs. 1 Mpps).

In Chapter 7, I devised a new version of the HIKe framework which I refer simply to as HIKe (Heal, Improve and desKill eBPF). HIKe is an evolution of HIKe-v0, but it is

much more powerful and easier to use, so it can be programmed even by inexperienced eBPF developers. The most important feature introduced by HIKe was the possibility to compose/chain programs together following a custom logic and not necessarily a linear one (as in case of HIKe-v0). This custom logic is encapsulated in HIKe Chains, which can be programmed in the C language. A HIKe Chain can make use of variables, branch conditions, loops and, most importantly, can invoke eBPF programs exploiting the well-known *function call* paradigm. HIKe Chains are compiled into bytecodes run by the HIKe Virtual Machine (HIKe VM), shipped with the HIKe framework. The novelty of HIKe lies in the fact that HIKe Chains do not have to be verified by the eBPF verifier, bringing significant advantages in terms of faster design and prototyping of complex network functions.

On top of HIKe, the eCLAT framework was developed with the aim of simplifying the management of the entire HIKe framework and, at the same time, providing a Python-like scripting language for coding HIKe Chains. eCLAT helps programmers that are not expert of eBPF and kernel programming (referred to as *Muggles*) in the creation of complex applications by composing/chaining eBPF network programs already realized by experts (*Wizards*). Thanks to eCLAT the Muggles do not have to use C-language to implement HIKe Chains (i.e. to chain programs) and do not have to know the internals of the HIKe framework or even the eBPF core. The overhead of the HIKe architecture has been evaluated in terms of computational overhead and performance degradation considering a typical scenario and a worst case. The HIKe performance is already better with respect to competing tools for Muggles (i.e. Netfilter base solution such as *IP set*). The performance penalty with respect to raw eBPF programming is already acceptable considering the huge gain in simplicity for some applications. I expect that further work on the HIKe framework will also reduce its gap with respect to raw eBPF.

Bibliography

[1] The 500 most powerful supercomputers use Linux. https://en.lffl.org/2016/11/linux-top500-super-computer.html.

[2] A. Mayer, P. Loreti, L. Bracciale, P. Lungaroni, S. Salsano, C. Filsfils. Performance monitoring with h^2: Hybrid kernel/ebpf data plane for srv6 based hybrid sdn". *Elsevier Computer Networks*, 185, 2021.

[3] A. AbdelSalam *et. al.* Implementation of Virtual Network Function Chaining through Segment Routing in a Linux-based NFV Infrastructure. In *IEEE NetSoft 2017, Bologna, Italy*, July 2017.

[4] A. Bashandy et al. Segment Routing with the MPLS data plane. RFC 8660, December 2019.

[5] A. Abdelsalam et al. SERA: SEgment Routing Aware Firewall for Service Function Chaining scenarios. In *IFIP Networking 2018*, pages 46–54. IEEE, May 2018.

[6] Abdulfatah AG Abushagur, Tan Saw Chin, Rizaludin Kaspin, Nazaruddin Omar, and Ahmad Tajuddin Samsudin. Hybrid software-defined network monitoring. In *International Conference on Internet and Distributed Computing Systems*, pages 234–247. Springer, 2019.

[7] Zaafar Ahmed, Muhammad Hamad Alizai, and Affan A. Syed. Inkev: In-kernel distributed network virtualization for dcn. *ACM SIGCOMM Computer Communication Review*, 46(3), jul 2018.

[8] Ian F Akyildiz, Ahyoung Lee, Pu Wang, Min Luo, and Wu Chou. A roadmap for traffic engineering in sdn-openflow networks. *Computer Networks*, 71:1–30, 2014.

[9] Zafar Ali, Clarence Filsfils, Satoru Matsushima, Daniel Voyer, and Mach Chen. Operations, Administration, and Maintenance (OAM) in Segment Routing Networks with IPv6 Data plane (SRv6). Internet-Draft draft-ali-6man-spring-srv6-oam-03, Internet Engineering Task Force, July 2019. Work in Progress.

[10] Engineering at Meta. Open-sourcing Katran, a scalable network load balancer, 2021.

[11] D. Barach et al. Batched packet processing for high-speed software data plane functions. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, 2018.

[12] David Barach, Leonardo Linguaglossa, Damjan Marion, Pierre Pfister, Salvatore Pontarelli, and Dario Rossi. High-speed software data plane via vectorized packet processing. *IEEE Communications Magazine*, 56(12):97–103, 2018.

[13] Bcc project. https://github.com/iovisor/bcc.

[14] T. Begin et al. An accurate and efficient modeling framework for the performance evaluation of DPDK-based virtual switches. *IEEE Transactions on Network and Service Management*, 2018.

[15] Gilberto Bertin. XDP in practice: integrating XDP into our DDoS mitigation pipeline. In *Technical Conference on Linux Networking, Netdev*, volume 2, pages 1–5, Nepean, Canada, 2017. The NetDev Society.

[16] Matteo Bertrone, Sebastiano Miano, Jianwen Pi, Fulvio Risso, and Massimo Tumolo. Toward an eBPF-based clone of iptables. In *Netdev 0x12, THE Technical Conference on Linux Networking*, Nepean, Canada, 2018. The NetDev Society.

[17] Matteo Bertrone, Sebastiano Miano, Fulvio Risso, and Massimo Tumolo. Accelerating linux security with ebpf iptables. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, pages 108–110, 2018.

[18] Gianluca Borello. The art of writing ebpf programs: a primer. `https://sysdig.com/blog/the-art-of-writing-ebpf-programs-a-primer`, 2019.

[19] Daniel Borkmann. Advanced programmability and recent updates with tc's cls bpf. *Proc. NetDev*, 1, 2016.

[20] S. Bradner and J. McQuaid. Benchmarking Methodology for Network Interconnect Devices. RFC 2544, RFC Editor, March 1999.

[21] S.O. Bradner. Benchmarking Terminology for Network Interconnection Devices. RFC 1242, July 1991.

[22] Frank Brockners, Shwetha Bhandari, and Tal Mizrahi. Data Fields for In-situ OAM. Internet-Draft draft-ietf-ippm-ioam-data-10, Internet Engineering Task Force, July 2020. Work in Progress.

[23] Gabriel Brown. Service Chaining in Carrier Networks. *Heavy Reading White Paper*, 2015.

[24] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. hxdp: Efficient software packet processing on FPGA NICs. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 973–990, 2020.

[25] C. Filsfils, D. Dukes (eds.) et al. IPv6 Segment Routing Header (SRH). RFC 8754, March 2020.

[26] C. Filsfils (ed.) et al. Compressed SRv6 Segment List Encoding in SRH. Internet-Draft draft-filsfilscheng-spring-srv6-srh-comp-sl-enc, Internet Engineering Task Force, May 2020. Work in Progress.

[27] C. Filsfils, et al. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986, February 2021.

[28] C. Filsfils, P. Camarillo (eds.) et al. Network Programming extension: SRv6 uSID instruction. Internet-Draft draft-filsfils-spring-net-pgm-extension-srv6-usid, Internet Engineering Task Force, May 2020. Work in Progress.

[29] Cyril Cassagnes, Lucian Trestioreanu, Clement Joly, and Radu State. The rise of ebpf for non-intrusive performance monitoring. In *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2020.

[30] Openstack celiometer. `https://docs.openstack.org/ceilometer/latest/`.

[31] C. Filsfils et al. SRv6 Network Programming. Internet-Draft draft-ietf-spring-srv6-network-programming, Internet Engineering Task Force, October 2020. Work in Progress.

[32] Tracy Yingying Cheng and Xiaohua Jia. Compressive traffic monitoring in hybrid sdn. *IEEE Journal on Selected Areas in Communications*, 36(12):2731–2743, 2018.

[33] CloudLab. `https://www.cloudlab.us/`.

[34] J. Corbet. Large receive offload. `https://lwn.net/Articles/243949/`.

[35] CSIT REPORT - The Fast Data I/O Project (FD.io) Continuous System Integration and Testing (CSIT) project report for CSIT master system testing of VPP-18.04 release. `https://docs.fd.io/csit/master/report/_static/archive/csit_master.pdf`.

[36] D. Barach, L. Linguaglossa, D. Marion, P. Pfister, S. Pontarelli, D. Rossi, and J. Tollet. Batched packet processing for high-speed software data plane functions. In *IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE INFOCOM 2018, 2018.

[37] D. Frost and S. Bryant. Packet Loss and Delay Measurement for MPLS Networks. IETF RFC 6374, September 2011.

[38] D. Kreutz, F. M. V. Ramos, P. E. Veríssimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. Software-defined networking: A comprehensive survey. *Proceedings of the IEEE*, 103:14–76, 2015.

[39] D. Mills, J. Martin. Network Time Protocol Version 4: Protocol and Algorithms Specification. IETF RFC 5905, June 2010.

[40] D. Dukes et al. SR for SDWAN - VPN with Underlay SLA. Internet-Draft draft-dukes-spring-sr-for-sdwan, Internet Engineering Task Force, June 2019. Work in Progress.

[41] Google Developers. Protocol Buffers. `https://developers.google.com/protocol-buffers`, 2021.

[42] DPDK. `https://www.dpdk.org/`.

[43] DPDK - Supported Hardware. https://core.dpdk.org/supported/.

[44] A thorough introduction to eBPF. https://lwn.net/Articles/740157/.

[45] BPF CO-RE (Compile Once – Run Everywhere). https://nakryiko.com/posts/bpf-portability-and-co-re/.

[46] bpf: Program extensions or dynamic re-linking. https://lore.kernel.org/bpf/9b204165-1d32-0e0d-ce19-10acaa45f9ed@fb.com/T/.

[47] bpf: Introduce global functions. https://lwn.net/ml/netdev/20200109063745.3154913-1-ast@kernel.org/.

[48] Linux kernel Git - commit: c04c0d2 bpf: increase complexity limit and maximum program size. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=c04c0d2b968ac45d6ef020316808ef6c82325a82.

[49] ebpf - instruction set architecture. https://www.kernel.org/doc/html/latest/bpf/instruction-set.html.

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

[51] ethtool - Linux man page. https://linux.die.net/man/8/ethtool.

[52] F. Clad and X. Xu (eds.) et al. Segment Routing for Service Chaining. Internet-Draft, December 2017.

[53] C. Filsfils et al. The Segment Routing architecture. In *Global Communications Conference (GLOBECOM), 2015 IEEE*, pages 1–6. IEEE, 2015.

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

[55] Clarence Filsfils et al. Interconnecting Millions Of Endpoints With Segment Routing, March 2018. Work in Progress.

[56] Giuseppe Fioccola, Tianran Zhou, Mauro Cociglio, and Fengwei Qin. IPv6 Application of the Alternate Marking Method. Internet-Draft draft-fz-6man-ipv6-alt-mark-09, Internet Engineering Task Force, May 2020. Work in Progress.

[57] Open Networking Foundation. Software-Defined Networking: The New Norm for Networks, 2012. available online at https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[58] Rakesh Gandhi, Clarence Filsfils, Daniel Voyer, Mach Chen, and Bart Janssens. Performance Measurement Using TWAMP Light for Segment Routing Networks. Internet-Draft draft-gandhi-spring-twamp-srpm-08, Internet Engineering Task Force, March 2020. Work in Progress.

[59] Rakesh Gandhi, Clarence Filsfils, Daniel Voyer, Stefano Salsano, and Mach Chen. Performance Measurement Using UDP Path for Segment Routing Networks. Internet-Draft draft-gandhi-spring-rfc6374-srpm-udp-03, Internet Engineering Task Force, November 2019. Work in Progress.

[60] E. Gershuni et al. Simple and Precise Static Analysis of Untrusted Linux Kernel Extensions. In *PLDI 2019: Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery.

[61] G. Fioccola, Ed. Alternate-Marking Method for Passive and Hybrid Performance Monitoring . IETF RFC 8321, January 2018.

[62] Yoann Ghigoff, Julien Sopena, Kahina Lazri, Antoine Blin, and Gilles Muller. Bmc: Accelerating memcached using safe in-kernel caching and pre-stack processing. In *NSDI*, pages 487–501, 2021.

[63] Ryan Gonzalez, Philip House, Ivan Levkivskyi, Lisa Roach, and Guido van Rossum. Python syntax for syntax for variable annotations. PEP 526, 2016.

[64] GRPC: A high performance, open-source universal RPC framework. https://grpc.io/.

[65] J. Halpern and C. Pignataro. Service Function Chaining (SFC) Architecture. IETF RFC 7665, October 2015.

[66] N. Handigol et al. Reproducible network experiments using container-based emulation. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 253–264. ACM, 2012.

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

[68] HIKe eCLAT Prototype Home Page. https://https://github.com/hike-eclat/docs/blob/master/docs/quickstart.rst.

[69] HIKe eCLAT Technical Documentation. https://hike-eclat.readthedocs.io/en/latest/index.html.

[70] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David 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*, pages 54–66, 2018.

[71] Jonghwan Hyun, Nguyen Van Tu, and James Won-Ki Hong. Towards knowledge-defined networking using in-band network telemetry. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7. IEEE, 2018.

[72] 1588-2008 IEEE Standard for a Precision Clock Synchronization Protocol for Networked Measurement and Control Systems. IEEE, 2008.

[73] IPset. http://ipset.netfilter.org.

[74] Arch Wiki - iptables. https://wiki.archlinux.org/index.php/iptables.

[75] A JIT for packet filters. https://lwn.net/Articles/437981/.

[76] Clement Joly and François Serman. Evaluation of Tail Call Costs in eBPF. In *Linux Plumbers Conference 2020*, page 14, San Francisco, California, 2020. The Linux Foundation.

[77] JSON-RPC. https://www.jsonrpc.org.

[78] K. Hedayat, R. Krzanowski, et al. A Two-Way Active Measurement Protocol (TWAMP). IETF RFC 5357, September 2006.

[79] Lightweight Tunnel. https://lwn.net/Articles/650778/.

[80] Jakub Kicinski and Nicolaas Viljoen. ebpf hardware offload to smartnics: cls bpf and xdp. *Proceedings of netdev*, 1, 2016.

[81] Kubernetes. http://kubernetes.io.

[82] Kubernetes playground. https://github.com/ferrarimarco/kubernetes-playground.

[83] Minseok Kwon, Krishna Prasad Neupane, John Marshall, and M Mustafa Rafique. Cuvpp: Filter-based longest prefix matching in software data planes. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 12–22. IEEE, 2020.

[84] B. Lantz et al. A network in a laptop: rapid prototyping for software-defined networks. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, page 19. ACM, 2010.

[85] David Lebrun. Leveraging IPv6 Segment Routing for Service Function Chaining. In *CoNEXT 2015 student workshop*, 2015.

[86] David Lebrun. Reaping the bene ts of ipv6 segment routing. *PhD thesis*, 2017.

[87] David Lebrun and Olivier Bonaventure. Implementing ipv6 segment routing in the linux kernel. In *Applied Networking Research Workshop 2017*, 2017.

[88] Joshua Levin and Theophilus A Benson. Viperprobe: Rethinking microservice observability with ebpf. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*, pages 1–8. IEEE, 2020.

[89] Linux in 2020. https://www.linux.com/news/linux-in-2020-27-8-million-lines-of-code-in-the-ke

[90] Linux kernel. https://www.kernel.org/.

[91] Linux kernel 5.11. https://kernelnewbies.org/Linux_5.11.

[92] Linux kernel 5.13. https://kernelnewbies.org/Linux_5.13.

[93] Linux kernel 5.14. https://kernelnewbies.org/Linux_5.14.

[94] Linux kernel 5.17. https://kernelnewbies.org/Linux_5.17.

[95] Linux kernel 5.9. https://kernelnewbies.org/Linux_5.9.

[96] Generic receive offload. https://lwn.net/Articles/358910/.

[97] Scaling in the Linux Networking Stack. https://kernel.org/doc/Documentation/networking/scaling.txt.

[98] SMP IRQ affinity. https://www.kernel.org/doc/Documentation/IRQ-affinity.txt.

[99] seg6: fix the iif in the IPv6 socket control block. https://patchwork.kernel.org/project/netdevbpf/patch/20211208195409.12169-1-andrea.mayer@uniroma2.it.

[100] seg6: fix skb transport_header after decap_and_validate. https://patchwork.ozlabs.org/project/netdev/patch/20191116150553.17497-3-andrea.mayer@uniroma2.it.

[101] seg6: fix srh pointer in get_srh(). https://patchwork.ozlabs.org/project/netdev/patch/20191116150553.17497-2-andrea.mayer@uniroma2.it.

[102] Linux Kmemleak. https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html.

[103] Linux Sysctl. https://www.kernel.org/doc/Documentation/sysctl/README.

[104] ftrace - Function Tracer. https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[105] 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)*, pages 408–413. IEEE, 2018.

[106] A. Mayer et al. An Efficient Linux Kernel Implementation of Service Function Chaining for legacy VNFs based on IPv6 Segment Routing. In *5th IEEE International Conference on Network Softwarization (NetSoft 2019)*. IEEE, 2019.

[107] Andrea Mayer. Hike vm - instruction set architecture. https://hike-eclat.readthedocs.io/en/latest/detailed_doc.html#supported-hike-vm-instructions, 2022.

[108] Steven McCanne and Van Jacobson. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, volume 46, 1993.

[109] S. Miano et al. A Framework for eBPF-Based Network Functions in an Era of Microservices. *IEEE Transactions on Network and Service Management*, 18(1):133 – 151, 2021.

[110] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Mauricio Vásquez Bernal, Yunsong Lu, Jianwen Pi, and Aasif Shaikh. A service-agnostic software framework for fast and efficient in-kernel network services. In *2019 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pages 1–9. IEEE, 2019.

[111] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. Creating complex network services with ebpf: Experience and lessons learned. In *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–8. IEEE, 2018.

[112] Gregory Mirsky, Guo Jun, Henrik Nydell, and Richard Foote. Simple Two-way Active Measurement Protocol. Internet-Draft draft-ietf-ippm-stamp-10, Internet Engineering Task Force, October 2019. Work in Progress.

[113] Tal Mizrahi, Gidi Navon, Giuseppe Fioccola, Mauro Cociglio, Mach Chen, and Greg Mirsky. Am-pm: Efficient network telemetry using alternate marking. *IEEE Network*, 33(4):155–161, 2019.

[114] M. Konstantynowicz et al. Multiple Loss Ratio Search for Packet Throughput (MLRsearch). Internet-Draft draft-vpolak-mkonstan-bmwg-mlrsearch-03, Internet Engineering Task Force, March 2020. Work in Progress.

[115] Al Morton. Active and Passive Metrics and Methods (with Hybrid Types In-Between). RFC 7799, May 2016.

[116] 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*, pages 67–72. ACM, 2018.

[117] Nagios it infrastructure monitoring. https://www.nagios.org/.

[118] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI) 20)*, pages 41–61, 2020.

[119] Linux Netfilter. http://www.netfilter.org.

[120] Network Function Framework for Go. https://github.com/intel-go/nff-go.

[121] ETSI Network Function Virtualization. http://www.etsi.org/technologies-clusters/technologies/nfv.

[122] Openflow switch specification. version 1.5.1. https://www.opennetworking.org/.

[123] Open vSwitch. available online at http://openvswitch.org.

[124] Open vSwitch with DPDK. http://docs.openvswitch.org/en/latest/intro/install/dpdk/.

[125] Federico Parola, Sebastiano Miano, and Fulvio Risso. A proof-of-concept 5g mobile gateway with ebpf. In *Proceedings of the SIGCOMM'20 Poster and Demo Sessions*, pages 68–69. 2020.

[126] N. Pitaev et al. Multi-VNF performance characterization for virtualized network functions. In *IEEE Conference on Network Softwarization (NetSoft)*, 2017.

[127] Cilium Project. Cilium project home page. https://cilium.io/, 2020. Accessed: 15-01-2021.

[128] LLVM project. The LLVM compiler infrastructure. https://llvm.org/.

[129] The Cilium project. BPF and XDP Reference Guide, 2021. available online at https://docs.cilium.io/en/latest/bpf/#bpf-and-xdp-reference-guide.

[130] Wander Queiroz, Miriam AM Capretz, and Mario Dantas. An approach for sdn traffic monitoring based on big data techniques. *Journal of Network and Computer Applications*, 131:28–39, 2019.

[131] P. Quinn et al. Network Service Header (NSH). Internet-Draft, November 2017.

[132] P. Quinn and T. Nadeau. Problem Statement for Service Function Chaining. IETF RFC 7498, April 2015.

[133] A. Abdelsalam et al. SRPerf: a Performance Evaluation Framework for IPv6 Segment Routing. *accepted to IEEE Transaction on Network and Service Management, arXiv preprint arXiv:2001.06182*, 2020.

[134] ETSI Group for NFV. Network Functions Virtualisation (NFV); Management and Orchestration; Functional requirements specification, 2016.

[135] P. Loreti et al. SRv6-PM: Performance Monitoring of SRv6 Networks with a Cloud-Native Architecture. *submitted to IEEE Transaction on Network and Service Management, arXiv preprint arXiv:2007.08633*, 2020.

[136] L. Rizzo. netmap: A novel framework for fast packet i/o. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 101–112. Boston, MA: USENIX Association, 2012. available online at https://www.usenix.org/conference/atc12/technical-sessions/presentation/rizzo.

[137] The rose project. https://netgroup.github.io/rose.

[138] Rami Rosen. Linux Kernel Networking - Implementation and Theory, 2014.

[139] S. Seth and M. A. Venkatesulu. Transmission and reception of packets. available online at https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470377833.ch18.

[140] S. Shalunov, B. Teitelbaum, et. al. A One-way Active Measurement Protocol (OWAMP). IETF RFC 4656, September 2006.

[141] Renato B Santos, Thiago R Ribeiro, and Cecília de AC César. A network monitor and controller using only openflow. In *2015 Latin American Network Operations and Management Symposium (LANOMS)*, pages 9–16. IEEE, 2015.

[142] S. Bryant, S. Sivabalan and S. Soni. UDP Return Path for Packet Loss and Delay Measurement for MPLS Networks. IETF RFC 7876, July 2016.

[143] Scapy - Packet crafting for Python2 and Python3.

[144] D. Scholz, D. Raumer, P. Emmerich, A. Kurtz, K. Lesiak, and G. Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 01, pages 209–217, 2018.

[145] SD-WAN. https://en.wikipedia.org/wiki/SD-WAN.

[146] A. Shaw. Multi-queue network interfaces with SMP on Linux. https://greenhost.nl/2013/04/10/multi-queue-network-interfaces-with-smp-on-linux/.

[147] S. Matsushima et al. SRv6 Implementation and Deployment Status. Internet-Draft draft-matsushima-spring-srv6-deployment-status, Internet Engineering Task Force, April 2020. Work in Progress.

[148] S. Previdi et al. Segment Routing Architecture. IETF RFC 8402, July 2018.

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

[150] SRv6 Control Plane in ROSE project. https://github.com/netgroup/rose-srv6-control-plane.

[151] Counters support for SRv6 behaviors (linux kernel). https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net.git/commit/?id=94604548aa71.

[152] Customizing creation/destruction of an SRv6 Endpoint behavior. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=cfdf64a03406.

[153] SRv6 End.DT4 behavior implementation in the Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/netdev/net-next.git/commit/?id=664d6f86868b.

[154] Support for vrftable attribute in SRv6 End.DT4/DT6 behaviors (iproute2). https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/commit/?id=69629b4e43c.

[155] SRv6 End.DT46 behavior implementation in the Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=8b532109bf88.

[156] SRv6 End.DT6 behavior (VRF mode) implementation in the Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=20a081b7984c.

[157] Counters support for SRv6 behaviors (iproute2). https://git.kernel.org/pub/scm/network/iproute2/iproute2.git/commit/?id=02ca3aabe92.

[158] SRv6-PM - SRv6 Performance Monitoring in Linux. https://netgroup.github.io/srv6-pm/, 2020.

[159] T. Szigeti, D. Zacks, M. Falkner, and S. Arena. Cisco Digital Network Architecture: Intent-based Networking for the Enterprise. *Addison-Wesley Professional Computing Series, volume 1*, Dec 2011.

[160] The eCLAT project. eCLAT docker Github Page. https://github.com/netgroup/eclat-docker, 2021.

[161] TRex realistic traffic generator. https://trex-tgn.cisco.com/.

[162] TRex Stateless Python API. https://trex-tgn.cisco.com/trex/doc/cp_stl_docs/index.html.

[163] Pang-Wei Tsai, Chun-Wei Tsai, Chia-Wei Hsu, and Chu-Sing Yang. Network monitoring in software-defined networking: A review. *IEEE Systems Journal*, 12(4):3958–3969, 2018.

[164] Cheng-Chun Tu, Joe Stringer, and Justin Pettit. Building an extensible open vswitch datapath. *ACM SIGOPS Operating Systems Review*, 51(1):72–77, 2017.

[165] William Tu et al. Bringing the Power of eBPF to Open vSwitch. In *Linux Plumbers Conference 2018*, page 11, San Francisco, California, 2018. The Linux Foundation.

[166] William Tu, Joe Stringer, Yifeng Sun, and Yi-Hung Wei. Bringing the power of ebpf to open vswitch. In *Linux Plumbers Conference*, 2018.

[167] uSID Linux Kernel implementation. https://netgroup.github.io/srv6-usid-linux-kernel/.

[168] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.

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

[170] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Building hybrid virtual network functions with express data path. In *2019 15th International Conference on Network and Service Management (CNSM)*, pages 1–9. IEEE, 2019.

[171] Nguyen Van Tu, Jae-Hyoung Yoo, and James Won-Ki Hong. Accelerating virtual network functions with fast-slow path architecture using express data path. *IEEE Transactions on Network and Service Management*, 17(3):1474–1486, 2020.

[172] Pier Luigi Ventre, Mohammad Mahdi Tajiki, Stefano Salsano, and Clarence Filsfils. SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Networks. *IEEE Transactions on Network and Service Management*, 15(4):1378–1392, 2018.

[173] P.L. Ventre et al. Segment Routing: A comprehensive survey of research activities, standardization efforts and implementation results. *accepted to IEEE Communications Surveys and Tutorials*, 2020. https://doi.org/10.1109/COMST.2020.3036826, preprint on arXiv: https://arxiv.org/pdf/1904.03471.

[174] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1–36, 2020.

[175] D. Vladislavić et al. Throughput Evaluation of Kernel based Packet Switching in a Multi-core System. In *International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2019.

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

[177] Virtual Routing and Forwarding (VRF). `https://www.kernel.org/doc/Documentation/networking/vrf.txt`.

[178] Strict mode for Virtual Routing and Forwarding (VRF) infrastructure. `https://lwn.net/Articles/823946/`.

[179] Prevent duplication of table id for VRF tables. `https://lore.kernel.org/netdev/20200307205916.15646-1-sharpd@cumulusnetworks.com/`.

[180] Shie-Yuan Wang and Jen-Chieh Chang. Design and implementation of an intrusion detection system by using extended bpf in the linux kernel. *Journal of Network and Computer Applications*, page 103283, 2021.

[181] W. Cheng et al. Shorter SRv6 SID Requirements. Internet-Draft draft-cheng-spring-shorter-srv6-sid-requirement, Internet Engineering Task Force, July 2020. Work in Progress.

[182] Mathieu Xhonneux. An interface for programmable ipv6 segment routing network functions in linux. *Maser thesis*, 2017-2018.

[183] Zuo Xiang, Frank Gabriel, Elena Urbano, Giang T Nguyen, Martin Reisslein, and Frank HP Fitzek. Reducing latency in virtual machines: Enabling tactile internet for human-machine co-working. *IEEE Journal on Selected Areas in Communications*, 37(5):1098–1116, 2019.

[184] Xtables-addons - additional extension to Xtables packet filter. `http://xtables-addons.sourceforge.net`.

[185] Herbert Xu. GSO: Generic Segmentation Offload. `https://lwn.net/Articles/188489/`.

[186] Zabbix moniotoring everuthing. `https://www.zabbix.com/`.

[187] ZeroMQ. `https://zeromq.org`.