UNIVERSIDADE FEDERAL DE MINAS GERAIS
Instituto de Ciências Exatas
Programa de Pós-Graduação em Ciência da Computação

Matheus S. Castanho

Chaining-Box: Uma Arquitetura de Encadeamento de Funções de Rede Transparente Usando BPF

Belo Horizonte
2020

Matheus S. Castanho

Chaining-Box: Uma Arquitetura de Encadeamento de Funções de Rede Transparente Usando BPF

Versão final

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação da Universidade Federal de Minas Gerais, como requisito parcial à obtenção do título de Mestre em Ciência da Computação.

Orientador: Marcos A. M. Vieira
Coorientadora: Cristina K. Dominicini

Belo Horizonte
2020

Matheus S. Castanho

Chaining-Box: A Transparent Service Function Chaining Architecture Leveraging BPF

Final version

Thesis presented to the Graduate Program in Computer Science of the Universidade Federal de Minas Gerais in partial fulfillment of the requirements for the degree of Master in Computer Science.

Advisor: Marcos A. M. Vieira
Co-Advisor: Cristina K. Dominicini

Belo Horizonte
2020

Castanho, Matheus S.

C346c Chaining-Box: A Transparent Service Function Chaining Architecture Leveraging BPF [manuscrito] / Matheus S. Castanho. — 2020.
xiv, 78 f.

Orientador(a): Marcos A. M. Vieira
Dissertação (mestrado) — Universidade Federal de Minas Gerais, Instituto de Ciências Exatas, Departamento de Ciência da Computação
Referências: f. 69-74.;

1. Computação. 2. Virtualização de funções de rede. 3. Redes programáveis 4. Encadeamento de funções de rede. I. Orientador. II. Coorientadora. III. Título.

CDU 519.6*22 (043)

Ficha catalográfica elaborada pela bibliotecária Belkiz Inez Rezende Costa CRB 6ª Região nº 1510

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Chaining-Box: A Transparent SFC Architecture Leveraging BPF

MATHEUS SALGUEIRO CASTANHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marcos Augusto Menezes Vieira - Orientador
Departamento de Ciência da Computação - UFMG

Cristina Klippel Dominicini

Profa. Cristina Klippel Dominicini - Coorientadora
Departamento de Informática - IFES

[illegible]

Prof. Daniel Fernandes Macedo
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Italo Fernando Scotá Cunha
Departamento de Ciência da Computação - UFMG

Magnos Martinello

Prof. Magnos Martinello
Departamento de Informática - UFES

Belo Horizonte, 27 de Março de 2020.

Acknowledgments

First of all I thank my Lord and Savior, Jesus Christ, for guiding me every single day of my life and teaching me how to depend more of Him. I could not have accomplished all this without the comfort and renewal given by His Word during the most difficult times.

My parents, Josafat and Rose, whose unconditional support, both emotionally and financially, have allowed me to go places I never imagined. My sister, Aline, for inspiring me with her life, faith and wisdom, inspiring me to be a better person and a better christian.

Sallie, my soon-to-be wife, who has been by my side through all my journey as a student. For being kind, gentle and helping me see things the way they really are. And also for bearing the separation and long-distance relationship needed so I could work on my Masters degree.

My advisors, Marcos and Cristina, for sharing their time and knowledge with me during the development of this work. I also thank Magnos Martinello, Moisés Ribeiro and Rodolfo Villaça, leading professors at NERDS lab at UFES, who had a tremendous impact on my research career on the years we worked together, and for also allowing me to use their infrastructure to develop parts of this work. Huge thanks also to Diego Mafioletti and Rodolfo Valentim for their time and effort to help me overcome challenges when prearing the experiments.

Finally, I thank Racyus Delano and my colleagues from Winet lab at UFMG, who were alongside me during many days, nights and weekends required to develop this thesis. The memories and friendships we built will for sure be remembered for years to come.

Resumo

O desenvolvimento e a adoção da Virtualização de Funções de Rede (NFV) permitiu grandes avanços na forma como serviços de rede são implantados e gerenciados. Isso possibilitou o projeto de infraestruturas mais capazes de responder a rápidas variações de demanda causada pela natureza dinâmica do tráfego de rede, ao mesmo tempo que reduziu o custo de manutenção.

Nesses cenários, há a necessidade de composição de diferentes funções de rede de maneira sequencial para criar serviços de rede mais complexos para operarem sobre o tráfego de entrada. Diversas abordagens para lidar com essa interconexão, também chamada de encadeamento de funções de rede (SFC), têm sido propostas nos últimos anos, incluindo implementações de alto desempenho usadas em ambientes de produção.

Porém, uma análise cuidadosa dessas propostas evidencia que a separação entre os planos de serviço e de dados não tem estado entre os principais objetivos de projeto. Como consequência, as arquiteturas de SFC existentes são feitas sob medida para ambientes e plataformas específicas, frequentemente dependendo de dispositivos de rede especializados ou modificados, seja em software ou hardware.

Este trabalho propõe uma nova arquitetura de SFC denominada Chaining-Box. Ela é baseada em uma ideia simples: integrar toda a funcionalidade necessária para habilitar SFC em cada função de rede na forma de estágios de processamento. Isso é feito de forma completamente transparente, exigindo pouco ou nenhum suporte da infrastrutura de rede e sem modificar as funções. Os estágios são implementados como programas BPF rodando dentro do kernel do Linux e provêm todas as ações de SFC enquanto os pacotes atravessam a pilha de rede.

Chaining-Box é descrita em detalhes, apresentando o seu projeto e comparando-a a outras implementações de SFC, destacando tanto suas vantagens quanto suas desvantagens. Uma análise experimental também é apresentada para demonstrar a aplicabilidade da arquitetura e o desempenho do protótipo implementado.

Palavras-chave: BPF, Virtualização de Funções de Rede, Redes Programáveis, Encadeamento de Funções de Rede.

Abstract

The development and adoption of Network Function Virtualization (NFV) has allowed great improvements in the way network services are deployed and managed by service providers. It has enabled the design of infrastructures more capable of responding to quick changes in demand due to the dynamic nature of traffic, while also providing reduction in management costs.

In this setting, it is often required to compose different Service Functions (SF) in a sequential manner to create complex network services that operate on incoming traffic. Several approaches to handle this interconnection, also called Service Function Chaining (SFC), have been proposed in recent years, including high-performance implementations used in production systems.

However, a careful analysis of these proposals shows that the separation of concerns between the service and data planes has not been among the main design objectives. As a consequence, the existing SFC architectures are tailor-made for specific environments and platforms, often relying on specialized or modified network devices, either in software or in hardware.

This work proposes a new SFC architecture called Chaining-Box. It is based on a simple idea: integrating all functionality needed to provide SFC into each SF as a set of processing stages. This is done in a fully transparent manner, requiring little to no support from the underlying infrastructure and without any modifications to functions. The stages are implemented as BPF programs running inside a Linux kernel to provide all SFC actions as the packets traverse the kernel stack.

Chaining-Box in detail, presenting its design and discussing how it compares to other existing SFC implementations, highlighting its advantages and also the disadvantages. An experimental analysis is also presented to demonstrate its applicability and the performance of the prototype implemented.

Palavras-chave: BPF, Network Function Virtualization, Programmable Networks, Service Function Chaining.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Many complex network environments rely on the execution of a sequence of several different applications, also called Service Functions (SF), to provide a complete service for each incoming packet. For example, requests may have to go through a firewall followed by a deep packet inspector (DPI) before accessing a database server. To fulfill this sequence, packets must be forced to go through each function before being sent to their final destination, an operation called Service Function Chaining (SFC).

In the past such functions used to be deployed as hardware middleboxes, and packets were steered through them using cumbersome and inflexible network configurations. This approach posed serious challenges for environments that must quickly respond to changes in demand, such as cloud and telecommunications datacenters [Quinn and Nadeau, 2015]. The introduction of the Network Function Virtualization (NFV) paradigm partially tackled these problems, migrating network functions from hardware middleboxes to a software-based model [Mijumbi et al., 2016].

Beyond the many challenges involved in this transition, new mechanisms were required to provide SFC for virtualized functions, beyond what traditional routing and switching could offer. Many organizations and research groups have worked towards enabling this feature [Halpern and Pignataro, 2015, Qazi et al., 2013a, Xhonneux et al., 2018], and many solutions are already in used production systems.

Early solutions used Software Defined Networking (SDN) techniques to provide function chaining [Zhang et al., 2013, Qazi et al., 2013a, Fayazbakhsh et al., 2014], often relying on OpenFlow-enabled switches to steer traffic through functions. With time, more holistic approaches were proposed, like the creation of high-performance frameworks for the development and interconnection of SFs [Martins et al., 2014, Panda et al., 2016, Zhang et al., 2016]. Today, SFC is provided by several mainstream platforms such as Open vSwitch (OVS) [OVS, 2020], VPP [VPP, 2020], OpenStack [Openstack, 2020] and OpenDaylight [OpenDaylight, 2020]. However, all these proposals share the same limitations: they are dependent on specialized network devices or require functions to be re-implemented using platform-specific constructs. These cause current SFC mechanisms to be tailor-made to the platforms or devices they were built for, and also tightly coupled to the underlying infrastructure.

Ideally, service functions and network infrastructure should be completely agnostic to any SFC mechanisms in place, allowing greater flexibility for modifications and reuse by different deployments, without having to re-invent the wheel. For such, a fully transparent SFC architecture would require (1) no knowledge by service functions (2) and network infrastructure, (3) while being generic enough to be deployed in different environments, facilitating portability. All this without abstaining from (4) the ability for quick reconfiguration and (5) from being easily scalable to multiple instances as current solutions. Thus the main goal of this work is to answer the question: *How to provide SFC in a fully transparent manner while meeting all these requirements?*

Those items translate directly into a set challenges, namely (1) how to operate *around* a service function, without its full cooperation, (2) how to use the network for packet steering without requiring changes to its operation, and (3) how to build the SFC mechanism upon a common denominator readily available to most platforms.

To solve these problems and answer the question above, this work proposes Chaining-Box, a new SFC architecture that fully decouples service and data planes, allowing functions and network infrastructure to be agnostic to the SFC functionality. It uses NSH [Quinn et al., 2018] to provide an overlay for SFC and condenses all SFC-related actions into a set of sequential stages executed as BPF [Miano et al., 2018] programs inside the standard Linux kernel. Each program is compiled to a bytecode and run by the kernel at several hook points, which are executed as packets flow through the system's network stack. This approach introduces a high level of independence between SFs, SFC architecture and the underlying network, allowing on-demand modifications to the architecture's components with little to no disruption to SFs.

The remainder of this text exposes concepts (§2), discusses further other solutions and their shortcomings (§3), presents a detailed view of the architecture design (§4) and the implementation of a prototype (§5). The results of an experimental evaluation to understand its performance and feasibility are presented (§6), and Chaining-Box's main benefits and drawbacks are also discussed (§7). The text is concluded with some final remarks and future work (§8). All the code developed is available on GitHub[1].

[1]https://github.com/mscastanho/chaining-box

Chapter 2

Background and Concepts

Before presenting Chaining-Box, some important background and concepts that form the basis of the architecture are presented and discussed in this chapter. Namely, a brief introduction to Service Function Chaining (SFC) including key IETF standards (§2.1), an original taxonomy of service function implementations (§2.2), details on Berkeley Packet Filter (BPF) technology (§2.3), an overview of the layers in the Linux networking stack (§2.4), and a brief introduction to container networking (§2.5).

2.1 Service Function Chaining (SFC)

Network Function Virtualization (NFV) is a topic that has received much attention from both academy and industry researchers in recent years. At its core, it consists on an approach to migrate applications that were traditionally implemented in hardware to virtualized infrastructures, running in commodity off-the-shelf devices. This decreases acquisition and management costs, while providing more flexibility and scalability [Mijumbi et al., 2016].

Network functions can be combined into chains, an ordered list of functions to be executed sequentially, providing composed services. Examples of such functions include firewalls, load balancers, and deep packet inspectors. This interconnection of functions is called Service Function Chaining (SFC).

Traditionally, SFC implementations relied on complex routing schemes to steer traffic through network functions [Quinn and Guichard, 2014]. This made modifications to service functions difficult, usually requiring significant changes in network configuration, which is error-prone and cumbersome. Because these deployments were tightly coupled to network topology, they also tended to be restricted to a specific Service Provider (SP) domain, hardly being applicable to different scenarios. Besides, different third-party service functions have a low level of interoperability.

SFC has gained momentum in recent years [Xia et al., 2015, John et al., 2013, Sahhaf et al., 2015, Kitada et al., 2014, Quinn and Guichard, 2014] causing the Internet Engineering Task Force (IETF) to publish a problem statement for SFC defining key areas that working groups could investigate towards new SFC solutions, in the form of RFC 7498 [Quinn and Nadeau, 2015]. This document states the key challenges faced by traditional service function chaining implementations and establishes focus areas to guide workgroups to propose new protocols and solutions for those issues.

2.1.1 IETF Reference SFC Architecture

A product of such effort is RFC 7665 [Halpern and Pignataro, 2015], a standard that proposes a reference architecture for SFC, illustrated in Figure 2.1. In this model, an SFC encapsulation header is added to incoming packets, containing information about the current chain being executed and the next hop in the sequence. The chaining is enabled by the interaction of four different types of elements:

- **Classifiers**: packets ingressing the SFC environment are classified to determine which chaining should be executed for them, following the insertion of the corresponding SFC encapsulation;
- **Service Functions**: execute some kind of processing over the packet. They can be divided into two categories: *SFC-aware* and *SFC-unaware*. The first is composed of functions that have knowledge about the SFC encapsulation and know how to operate on it. The second, also called legacy functions, are those not aware of the protocol, needing the cooperation of a Proxy to be part of the SFC environment;
- **Proxies**: are responsible for implementing an interface between SFC unaware functions and the SFC domain. They remove and re-insert the SFC encapsulation before and after the service function has executed, respectively;
- **Forwarders**: at each step in the chain, a forwarder is responsible for determining which is the next service to be executed in the chain. This decision is based on an SFC forwarding table configured by the administrator and on the fields of the packet's SFC encapsulation. The forwarder is also responsible for changing the external transport encapsulation to ensure the underlay can deliver the packet to the next function in the chain.

As shown in Fig 2.1, packets are classified upon arrival to the network by the Classifier. In this step a packet is matched against a set of pre-configured rules and a

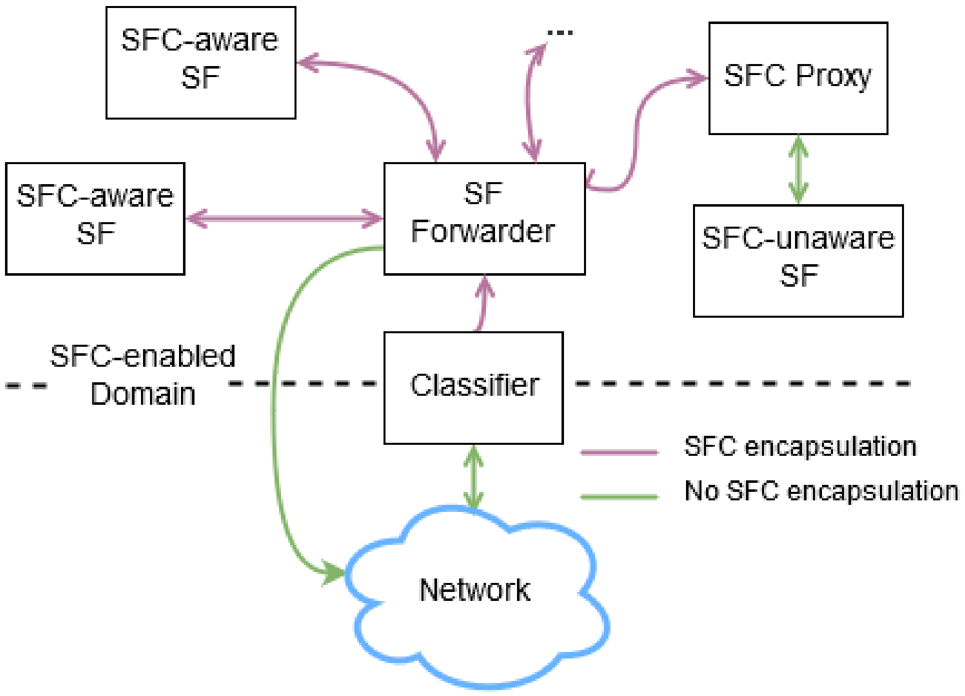


Figure 2.1: Illustration of RFC 7665 reference architecture. Adapted from [Halpern and Pignataro, 2015].

service path is chosen, if one exists for that packet. The SFC encapsulation containing the corresponding path information is added and the packet is sent to the next element in the architecture: the Forwarder. This element decides which SF should be executed next and alters the packet's external transport encapsulation accordingly, letting the network route it to the next hop or to its corresponding proxy, in case of an SFC-unaware function.

In that case, the Proxy will remove the SFC encapsulation and then forward the packet to the SF. After performing any operations on it, the SF will send the packet back to the Proxy for reinsertion and update of the SFC encapsulation, sending it back to the Forwarder afterwards. This process is repeated until the last SF in the chain has executed. That is when the Forwarder finally removes the SFC encapsulation and lets the packet follow its regular flow in the network.

2.1.2 Network Service Header (NSH)

RFC 8300 [Quinn et al., 2018] complements the architecture presented above with the specification of the Network Service Header (NSH), an encapsulation protocol to be used by SFC deployments. Combined, these standards are capable of providing service chaining in practice.

The NSH is independent of the outter transport encapsulation, but needs it to move the packets around the network. It provides service chaining as defined by RFC 7665 by adding an extra header to the packet. This new header is added between the outter transport encapsulation and the packet itself, as illustrated by Figure 2.2.

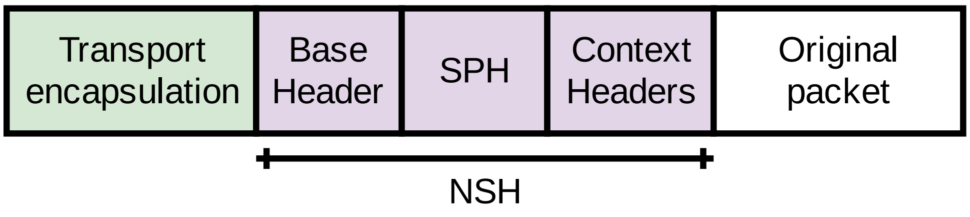


Figure 2.2: Packet encapsulated with NSH

The NSH header is divided into three parts:

- Base Header (4 Bytes): Provides general information like protocol version, next protocol, time to live (TTL), header length and MD Type, which determines the kind of metadata being carried.
- Service Path Header (4 Bytes): Holds path identification and current position within the service path.
- Context Header (Variable Length): Carries metadata to be shared between SFs.

A more detailed explanation of each field in the Base Header, the different MD Types available and metadata definition are out of the scope of this document. For such, please refer to the text of RFC 8300. Only MD Type 1 NSH was used throughout this work, which consists on four 4-byte fixed fields, 16 Bytes in total.

The Service Path Header (SPH) is a 4-byte field that consists of two subfields: a 3-byte Service Path Identifier (SPI) and an 1-byte Service Index (SI). The SPI represents an unique identifier for a service function path. This value is first set by the Classifier and is used by the other elements to know which service path is being executed on the packet. The SI is an index indicating the current location in the service function graph. The Classifier initially sets this value to 255, indicating the beginning of a path.

Throughout service path execution, if there is no packet reclassification, the SPI remains the same until the end. However, the SI is decremented by one after being processed by each service function. For SFC-aware SFs, this is done by the function itself, while for the SFC-unaware this is left to the Proxy.

To make the decision of what is the next hop for a given node in the graph, the Forwarder needs to maintain a lookup table with the SPI/SI pair mappings to service functions reachable from it. This table may also contain their respective addressing, along with the transport encapsulation to be used. A partial representation of a possible implementation of such table is illustrated in Table 2.1:

Table 2.1: Example of Forwarder's lookup table. Adapted from [Quinn et al., 2018].

SPI	SI	Next hop	Transport Encapsulation
10	255	192.0.2.1	VXLAN-gpe
10	254	198.51.100.10	GRE
10	251	198.51.100.15	GRE
40	251	198.51.100.15	GRE
50	200	01:23:45:67:89:ab	Ethernet
⋮	⋮	⋮	⋮
10	212	Null (end of path)	None

Using the table above as an example, suppose the Forwarder receives a packet with SPI=10 and SI=255. The Forwarder will check its table and know the next hop is at IP 192.0.2.1 through VXLAN-gpe. Once the packet is processed by that node it will have its SI value decremented and will be sent back to the Forwarder, which in turn will check the table again. This process is repeated for every service function in the chain, until the SI becomes 212. In this case the packet has reached the end of the path, indicated by the Null entry in the table. The packet will then be decapsulated and sent back to the network.

2.2 Service Function Types

The implementation of software service functions can be done in different ways, which alter the way they interface with the underlying operating system for packet processing. An original taxonomy is provided below in order to distinguish the different natures of implementations available and to clarify how Chaining-Box handles each one. The following types have been identified as being the most prominent:

2.2.1 Kernel-supported (KS)

These are functions implemented as common processes running in user space. They rely on standard mechanisms to receive packets from the network, such as system calls. This kind of implementation heavily relies on the facilities offered by the underlying operating system, which by default implements a complex network stack to handle multiple protocol types. However, with time this form of implementation has struggled to handle the ever-growing traffic rates supported by modern network cards. The penalty imposed by system interruptions and the constant switch between user and kernel modes for packet handling has hindered the use of this kind of applications for fast packet processing .

2.2.2 Kernel-bypass (KB)

As the name implies, applications implemented with this technique perform a complete bypass of the kernel stack and implement all packet processing functionality in user space. They usually make use of specialized drivers in polling mode so packets can be shared directly between network devices and user processes while avoiding additional delays caused by the traditional packet handling model based on interruptions and system calls. As a side effect, they have little to no dependence of the network facilities offered by the kernel stack, so all needed packet parsing and protocol handling has to be re-implemented in user space. Good examples of applications using this approach are those based on libraries such as DPDK [DPDK, 2020] and NetMap [Luigi, 2020].

2.2.3 Kernel-only (KO)

This class of applications aims to offer superior performance by using the opposite approach of KB programs: residing entirely in kernel space. Since one of the biggest performance bottlenecks in packet processing is the constant crossing of the barrier between kernel and user spaces, kernel-only functions implement all their functionality inside the kernel.

Classic examples are Click-based applications [Kohler et al., 2000] as well as packet processing leveraging *iptables*, both of which rely on the use of kernel modules, but have not been able to keep up with higher packet rates. KO functions have been gaining more traction in recent years [Miano et al., 2018] with the latest improvements to the BPF mechanism inside the Linux kernel (§2.3) and new technologies such as XDP (§2.4.1). These new technologies allow generic user-specified programs to be loaded to the kernel on run time, effectively allowing them to alter the kernel's network stack functionality in a dynamic manner.

2.3 Berkeley Packet Filter (BPF)

2.3.1 History

The Berkeley Packet Filter (BPF) [McCanne and Jacobson, 1993] was initially proposed by Steven McCanne and Van Jacobson in 1992 as a solution to perform packet filtering on the kernel of Unix BSD systems. It consisted of an instruction set and a virtual machine (VM) for executing programs written in that language.

The bytecode of a BPF application was transferred from user space to the kernel, where it was checked to assure security and prevent kernel crashes. After passing the verification, the system attached the program to a socket with an associated BPF VM and ran it on each arriving packet. The Linux kernel has supported BPF since version 2.5, with no major changes to the BPF core code until 2011, when besides the interpreter, the kernel gained support for a dynamic BPF translator [Dumazet, 2011]. Instead of interpreting the BPF byte code, the kernel was now able to translate BPF programs directly into x86 instructions.

One of the most prominent tools that use BPF is the `libpcap` library, used by the `tcpdump` tool. When using `tcpdump` to capture packets, a user can set a packet filtering expression so that only packets matching that expression are actually captured. For example, the expression `"ip and tcp"` captures all IPv4 packets that contain the TCP transport layer protocol. This expression can be reduced by a compiler to BPF bytecode.

2.3.2 Extended Berkeley Packet Filter (eBPF)

In 2013, BPF received a major upgrade with a whole set of new functionalities[1]. This new proposal was made by Alexei Starovoitov and was coined *extended Berkeley Packet Filter* (eBPF). With time, eBPF became the new standard and inherited the name of its predecessor, usually being referred to as BPF, with the previous version now being called *classic BPF* (cBPF). On the remaining sections in this text, the term BPF refers to the improved version used today (eBPF).

The differences between cBPF and eBPF are considerable, starting from the number of registers available, which increased from 2 to 11. Programs can now be chained

[1]https://lwn.net/Articles/740157/

in sequence using *tail calls*, and they have access to a whole new framework of helper functions and tools to interact with user space and many subsystems inside the Linux kernel. However, one of the most important is the addition of maps: key-value structures used to store data between program executions that can also be used to exchange data between programs and process in user space, for example. The kernel currently provides 20+ different types of maps, ranging from hash tables, arrays, stacks, queues, redirection tables, among others.

These new features greatly improved the applicability of BPF for the implementation of generic packet processing and tracing inside the kernel. This is proven by the growing adoption by major companies such as Facebook, Cloudflare, and Netronome [Facebook, 2018, Bertin, 2017, Beckett et al., 2018], which have already used BPF to implement network monitoring, network traffic manipulation, load balancing, and system profiling.

2.3.3 BPF system

The BPF system is composed of a series of components to compile, verify, and execute the source code of developed applications. The typical workflow of the BPF system is illustrated in Figure 2.3. It is written in a high-level language, mainly a subset of C, and compiled[2] to an ELF/object code containing the BPF instructions.

This file is then passed to a loader that can then insert it into the kernel using a special system call. During this process, the verifier analyzes the program and upon approval the kernel performs the dynamic translation (JIT). Besides being executed by the processor, the program can also be offloaded to specialized hardware.

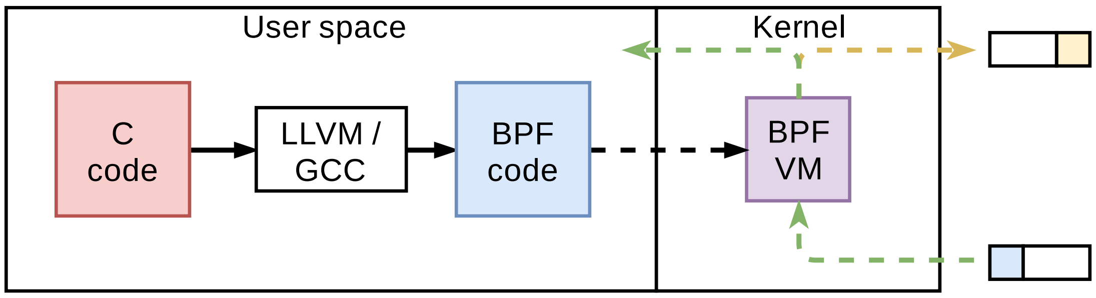


Figure 2.3: Typical workflow of loading BPF programs to the kernel.

To ensure the integrity and security of the operating system, the kernel uses a verifier that performs static program analysis of BPF instructions being loaded into the

[2]Both LLVM and GCC provide backends for BPF, but LLVM currently has better support as it was the compiler infrastructure adopted by the kernel community to implement most BPF-related features.

system. Among other things, it checks if a program is larger than maximum limit allowed (current limit is 10^6 instructions), whether or not the program terminates, if the memory accesses are within the memory range allowed for the program, and how deep the execution path is. It is called after the code has been compiled and during the process of loading the program into the data plane [Miller, 2017].

2.4 Linux Networking Stack

On Linux, packets entering the OS are processed by several layers in the kernel, as shown in Figure 2.4. They can be roughly divided in: socket layer, TCP stack, Netfilter, Traffic Control (TC), the eXpress Data Path (XDP), and the NIC.

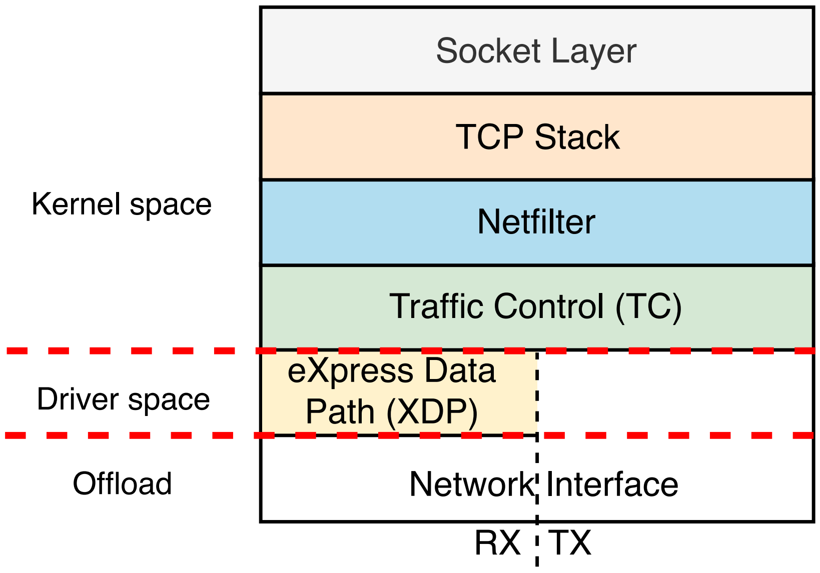


Figure 2.4: Linux kernel network stack.

Packets destined to a standard user space application go through all these layers and can be intercepted and modified during this process by modules such as `iptables`, which resides in the Netfilter layer. As explained before, BPF programs can be attached to several places inside the kernel, enabling packet mangling and filtering.

BPF programs attached to each layer see different contexts, i.e. the input data passed to them, and have distinct sets of helper functions available for use. For example, programs in the XDP layer are the first to interact to incoming packets on RX and and received a Layer-2 frame as its context. Moreover, they can call specific helper functions only available in this layer, for altering a packet's size, for example. Programs on the Netfilter layer, for example, see a Layer-3 packet as their context, not being able to alter the Ethernet header, but they have extra helper functions related to routing, checksum calculation and tunneling, not available on the XDP. In the following subsections the two main layers used in this work are described: XDP (§2.4.1) and TC (§2.4.2).

2.4.1 eXpress Data Path (XDP)

XDP is the lowest layer of the Linux kernel network stack. It is present only on the RX path, inside a device's network driver, allowing packet processing at the earliest point in the network stack, even before memory allocation is done by the OS. It exposes a hook to which BPF programs can be attached and executed for every received packet [Høiland-Jørgensen et al., 2018].

In this hook, programs are capable of taking quick decisions about incoming packets and also performing arbitrary modifications on them, avoiding additional overhead imposed by processing inside the kernel. This renders the XDP as the best hook in terms of performance speed for applications such as mitigation of DDoS attacks.

After processing a packet, an XDP program returns an action, a value that represents the final verdict regarding what should be done to the packet after program exit. The possible actions include dropping, passing it along the stack, sending it back to the network or forwarding it to another interface.

XDP is designed for fast packet processing applications while also improving programmability. In addition, it is possible to add or modify these programs without modifying the kernel source code, just like with other BPF hooks. For extra performance, programs can also be offloaded to compatible SmartNICs to be executed in hardware.

This layer can also be used to add programmability to kernel-bypass applications that use AF_XDP sockets [3]. This special socket family was designed to provide the benefits of the XDP to applications that do not make use of the kernel's network facilities, but do all its execution in user space, such as those based on the DPDK set of libraries. With them, XDP programs are executed before packets are sent to user space, combining the the programmability offered by XDP with the extra performance provided by kernel-bypassing.

[3]https://lwn.net/Articles/750293/

2.4.2 Traffic Control (TC) Hook

Although the XDP layer is well suited for many interesting applications, it can only process ingress traffic. The closest layer to the NIC on egress is the Traffic Control (TC) layer, also available on ingress. It is responsible for executing traffic control policies on Linux. In it, network administrators can configure different queuing disciplines (*qdisc*) for the various packet queues present in the system, as well as add filters to deny or modify packets.

The TC has a special queuing discipline type called `clsact` that exposes a hook that allows queue processing actions to be defined by BPF programs. They receive pointers to Ethernet frames like on XDP, but when a packet reaches the TC on ingress, the kernel has already parsed its headers to extract protocol metadata, hence richer context information is passed to the BPF programs attached to it. Extra helper functions are also available for programs on this layer compared to XDP.

2.5 Container networking

Containers are being widely used today to deploy cloud applications aa they provide a lightweight and reproducible environment that can be used both during development as in production. Tools like Docker [Merkel, 2014] and Kubernetes [Burns et al., 2016] have become commonplace and are the de facto standard for application deployment. Although a deeper discussion about container platforms is out of the scope if this text, the way Docker manages container networking is briefly discussed, as it was the main container technology used the main container technology used in this work.

A container is a process supported by several additional mechanisms to provide isolation between instances. These aim to guarantee separation in terms of networking, file system management, memory access, and so on. When it comes to networking, Docker provides isolation by using *namespaces* on Linux. Each namespace represents a separate network environment with its own set of interfaces, routing tables and configuration, and usually each container has its own network namespace. Although several ways to provide connectivity between containers are supported by Docker [Marmol et al., 2015] one of the most common is a combination of *veth pairs* and bridges.

Veth (virtual Ethernet) interfaces are always created in pairs and represent a direct link implemented in software: every packet transmitted on one side will be received by the other. The `veth` interface type received support for Native XDP on kernel 4.14 [Linux, 2017]. This kind of device can be used to create a direct connection between two containers or, more commonly, to create a link between the container and its host. This is done by attaching one veth interface on the container's network namespace and the other on the host's namespace. However, it is usually necessary to provide communication between containers as well, in which case a bridge or virtual switch is used. In this case, the end of the pair given to the host is connected to a virtual switch, like a Linux bridge or Open vSwitch (OVS) [Pfaff et al., 2015], which forwards packets between containers.

Chapter 3

Related work

This chapter presents a discussion about how Chaining-Box relates to other existing SFC mechanisms. In summary, decoupling service and data planes has not been a primary objective until now. And even when it has, the final implementations fall short on this feature. Here Chaining-Box is compared to early SDN- (§3.1), platform- (§3.2), and NSH-based approaches (§3.3), as well as others based on segment routing (§3.4). Table 3.1 presents a summary of the differences between Chaining-Box and the other proposals discussed. Finally, other projects leveraging BPF in different ways are present to show how this technology is being used today (§3.5).

3.1 SDN-based

Most of the initial SFC solutions relied on Software Defined Networking (SDN) to steer packets through chains [Medhat et al., 2017]. For example, SIMPLE [Qazi et al., 2013b] used a combination of SDN rules and tags to keep track of the current state of execution of a chain. Middleboxes were treated as blackboxes and all SFC functionality was implemented by network devices. Flowtags [Fayazbakhsh et al., 2014], on the other hand, modified middleboxes to add context-related tags, which in turn were used by switches to steer packets. Both solutions are heavily coupled to the network, as they rely on SDN-enabled switches to operate on specific tags in order to realize the chains.

3.2 Platform-based

Another set of works propose complete platforms for NFV development and execution, including function interconnection. These cover a wider scope than Chaining-Box, but are mentioned here as they provide innovative ways to implement chaining functionality.

NetBricks [Panda et al., 2016] provides a framework for NFV which is capable of running functions implemented using a custom language. As all functions reside in the same execution environment, chaining is provided in terms of function calls between SFs, leading to low overhead. A somewhat similar approach is taken by OpenNetVM [Zhang et al., 2016], an NFV platform that creates an abstraction layer on top of DPDK, through which network applications can be quickly developed using the provided constructs. It uses an internal shared memory mechanism to move packets between functions at low cost. Due to their holistic approach, these proposals rely on internal infrastructure provided by the surrounding platforms, and are not suitable as generic, function-agnostic architectures.

Polycube [Miano et al., 2019] is another software framework that allows the creation of generic service functions, this time based on BPF. Several helpers are provided to facilitate the development of services, which are split in a fast path running as BPF programs inside the kernel an a slow path in user space for management tasks. Just as Chaining-Box, it also benefits from the advantages offered by BPF technology such as low overhead and the ability to load programs on demand, but like the other frameworks discussed requires the re-implementation of functions for this environment.

The chaining mechanism used by OpenStack [Openstack, 2020] is also worth noting. OpenStack is a modular software that aims to provide management for computing, storage and networking resources across a pool of individual servers. It is widely used to build private clouds and provide Infrastructure as a Service (IaaS). It is implemented in a distributed manner, being composed by several separate modules acting together, each providing different types of services.

Neutron is the module responsible for networking. It manages the creation of networks, routes packets between VMs, handles access to the external world, takes care of access control, etc. Every computing instance is attached to a virtual network through Neutron ports. This provides a convenient way to handle service function chaining as a service path is expressed by a list of pairs of ingress and egress Neutron ports. Service paths also have flow classifiers, which decide what packets should be handled by each corresponding path. Neutron uses this information to send packets from egress to ingress ports, following the sequence in the port pair list, thus providing service function chaining. Since this approach heavily relies on Neutron's inner constructs, it is limited to the

OpenStack platform.

3.3 NSH-based

Some implementations also rely on the NSH protocol to implement SFC. Standalone Open vSwitch (OVS) [OVS, 2020] provides support for encapsulation, decapsulation and forwarding based on NSH using extensions to the OpenFlow standard. SFC can be fully implemented using OVS as the virtual switch to connect service functions.

OpenDaylight [OpenDaylight, 2020] features among the most widely used SDN controllers nowadays. Its SFC implementation is based on RFC 7665 and uses NSH as the SFC encapsulation of choice. SFC element functionality is integrated into other components, instead of being standalone entities. Classifiers are implemented in two ways: using OpenFlow switches such as OVS or using ip(6)tables with NetFilterQueue. During chain creation rules are added to OVS switches that will act as Forwarders and steer packets through the network functions. SFC Proxies are treated as abstract entities, capable of being implemented as an OVS switch, a VNF or even a physical network function.

Fast Data (FD.io) [FDio, 2020] is a Linux Foundation's project whose objective is to build an universal data plane, aiming for speed, efficiency, flexibility and scalability. It is a collection of projects and libraries to support and improve software-based packet processing. One of its core projects is the Vector Packet Processing (VPP) library which was open sourced by Cisco. This library enables packet processing using a graph abstraction, allowing developers to create new graph nodes and users to compose processing graphs built to their needs. NSH SFC is one the projects under FD.io's umbrella, it seeks to create the mechanisms necessary to support service chaining using NSH on VPP. The implementation is also based on RFC 7665 and NSH, and implements each SFC element as VPP graph nodes.

These projects are also coupled to the network as their implementation of SFC logical elements resides in network devices used by the network, thus requires infrastructure support. Chaining-Box is capable of fully replacing these complex mechanisms when it comes to providing SFC, enabling the same functionality even on top of simpler environments, e.g. based on Linux bridges and others that do not natively support SFC.

Previous work by the author also aimed to decouple service and data planes [Castanho et al., 2018]. PhantomSFC turns each RFC 7665 element into separate service functions, instead of logical roles played by network devices. These special SFs run alongside other functions, but have the sole objective of offering services needed to en-

Name	Type	Unchanged functions?	Platform-agnostic?	Network-agnostic?	Generic logical topology?
SIMPLE	SDN	Yes	No	No	Yes
FlowTags	SDN	No	No	No	Yes
NetBricks	Platform	No	No	Yes	Yes
OpenNetVM	Platform	No	No	Yes	Yes
Polycube	Platform	No	Yes	Yes	Yes
OpenStack	Platform	Yes	No	Yes	Yes
Open vSwitch	NSH	Yes	Yes	No	Yes
OpenDaylight	NSH	Yes	No	No	Yes
VPP	NSH	Yes	Yes	No	Yes
PhantomSFC	NSH	Yes	Yes	Yes	No
Segment Routing	SR	No	No	No	Yes
Chaining-Box	NSH	Yes	Yes	Yes	Yes

Table 3.1: Comparison between related SFC mechanisms and Chaining-Box

able SFC (classification, proxying and forwarding). As every packet needs to go through a Forwarder at each step of a chain, the logical topology has several star-like clusters centered on Forwarder SFs, which become significant points of failure and performance bottlenecks. The same problem applies to Proxies, at a lesser degree. Chaining-Box solves these problems by splitting SFC operations into all SFs, removing the star topology and the bottleneck of separate Forwarders.

3.4 Segment Routing

A new technique that has been used recently to enable SFC is segment routing (SR) using IPv6 [Abdelsalam et al., 2017, Duchene et al., 2018]. In this approach, each packet is encapsulated with a Segment Routing Header (SRH) [Filsfils et al., 2019], an extension to the IPv6 protocol, containing a stack of IPv6 addresses of nodes in a chain, called segments. At each hop, SR-enabled network devices execute their correspondent service functions and pop the top-most address from the stack, rewriting the destination IPv6 address to the next segment in the chain. Intermediate SR-unaware devices simply operate on the regular IPv6 fields to route the packet normally.

In this approach, the entire sequence of instances to operate over the packet is rendered by an initial classifier, simplifying the intermediate forwarding elements. On the other hand, it can only be used with IPv6 or MPLS protocols, limiting its scope. In addition, the extra packet size needed to specify the list of addresses can be big for long chains, increasing packet size and overall header overhead. Xhonneaux *et al.*

[Xhonneux et al., 2018] propose an SFC scheme using BPF programs in the Linux kernel stack to implement segment routing. It resembles closely the approach taken by Chaining-Box, with the difference being how the the underlying SFC technique used.

3.5 Projects leveraging BPF

Just like BPF is the key technology behind Chaining-Box, it has also enabled the development of many other relevant projects in recent years. InKev [Ahmed et al., 2018] enables to execute BPF programs on the datapath for virtual networks, targeting data center networks. Tu *et al.* [Tu et al., 2017] describe the design, implementation and evaluation of a BPF-based extensible datapath for OVS. To enable OpenFlow to parse arbitrary field, Jouet *et al.* [Jouet et al., 2015] defined an OpenFlow Extended match filed (OXM) to install cBPF bytecode and added a libpcap engine to Openflow software switch to execute it [Jouet et al., 2015]. Xhonneux *et al.* [Xhonneux et al., 2018] utilizes BPF to provide a programmable interface for IPv6 Segment routing, as discussed previously.

BPF is also being used in production systems. Examples include Cloudflare, which uses programs in multiple hooks in its network stack to implement DDoS mitigation, load balancing, and socket filtering and dispatching [Marek Majkowski, 2019]. Facebook implemented and open-sourced an L4 load balancer based on XDP, called Katran [Facebook, 2018]. Beyond the networking field, BPF has also proven an invaluable tool for tracing, the reason why it is being used by Netflix for performance monitoring and system profiling [Koch et al., 2019].

Moreover, the Cilium [Cilium, 2019] open-source project uses BPF extensively to provide networking and security for microservice applications, being aware of API-level details beyond simple network headers. Weave Scope leverages BPF to track TCP connections on Kubernetes clusters [WeaveWorks, 2017], and Project Calico has also announced recenty that a new data plane for container networking based on eBPF is being developed [Pollitt, 2019].

Chapter 4

Chaining-Box

As discussed before, existing SFC architectures are usually dependent on specialized devices or limited to specific environments, making them tailor-made for a single environment and set of technologies. In order to create a more generic chaining architecture, one that is not coupled to the network and can be applied to different scenarios, it is necessary to move all SFC functionality from the standard data plane to a separate service plane. This idea resembles more closely what was initially envisioned by RFC 7665 and removes the need for underlay network devices to have knowledge about the SFC protocols and mechanisms in use.

This chapter presents Chaining-Box, an SFC architecture that aims to provide chaining functionality in a transparent manner, requiring little cooperation from the underlay network besides forwarding packets based on standard Internet protocols. In the following sections, its data (§4.1) and control planes (§4.2) are discussed, followed by an architecture overview (§4.3) and a comparison with other approaches (§4.4).

4.1 Data Plane

Inspired by RFC 7665, Chaining-Box uses an extra SFC encapsulation to provide the chaining, namely the NSH. The operations over this header are limited to only three kinds: classification, insertion/removal, and forwarding of packets between nodes in a chain. To fully decouple the service plane from the data plane, Chaining-Box uses a simple approach: incorporate all SFC-related actions into the execution unit of each service function, i.e. virtual machines or containers.

Ideally, the addition of such features should be done without the need to refactor or modify the source code of a service function, being totally transparent to it. This can be done in the form of processing stages run before and after the SF's execution, as illustrated by Figure 4.1. Together with the function per se, these stages form a single box fully capable of forwarding packets between different SFs organized in chains, hence the name of the architecture. Individually, each of these boxes is called a *CBox*.

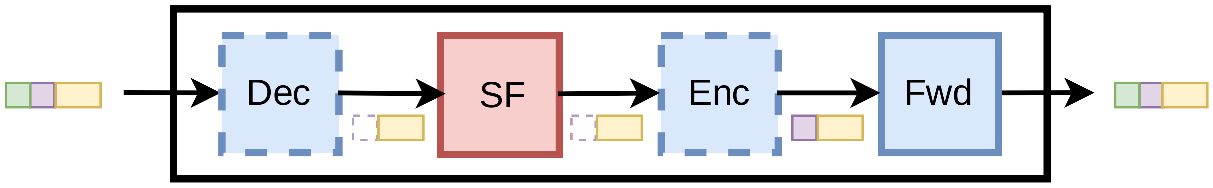


Figure 4.1: Example of a CBox and its processing stages

As shown above, a CBox is composed of four separate processing stages:

1. **Decapsulation (*Dec*)**: upon reception of an NSH-encapsulated packet, this stage is responsible for removing the NSH header and storing this information temporarily for later header re-insertion by the encapsulation stage. In case of an NSH-aware function, this stage is not needed;

2. **Service Function (*SF*)**: this stage represents the actual service function, and is not directly implemented by the architecture, but packets must be steered through it after the decapsulation stage;

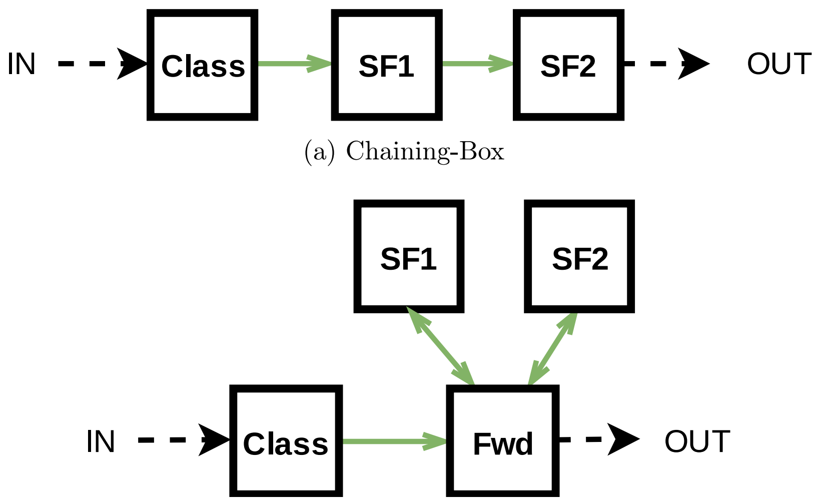
3. **Encapsulation (*Enc*)**: after the packet has been processed by the service function, the NSH is re-inserted by this stage, also decrementing the SI field. In case of an NSH-aware function, this stage is not needed;

4. **Forwarding (*Fwd*)**: this stage makes forwarding decisions based on the SI and SPI indexes from the NSH header. It looks up an SFC forwarding table and rewrites the packet's transport encapsulation headers before sending it back to the network to be forwarded to the next hop in the chain.

Dec and Enc stages implement all proxy actions, so they are only needed when dealing with an NSH-unaware SF. For NSH-aware functions, however, no changes on the SFC encapsulation are necessary, and thus these stages can be deactivated, hence the dashed lines on Figure 4.1. In this case, packets sent to a CBox are directly received by the SF stage. Since the function is agnostic of the processing stages, Chaining-Box supports different kinds of packet matching algorithms to remove and re-insert the NSH, only requiring modifications to how Dec and Enc stage are implemented.

One of the greatest benefits of local 1-to-1 forwarding stages instead of a centralized 1-to-many forwarding element as in RFC 7665 is that it allows functions to communicate directly, without the need of an intermediary, greatly reducing the overhead on forwarding components and avoiding single points of failure. This allows the architecture to assume more generic logical topologies and a more distributed form, less centralized on individual elements. This difference in logical topology can be observed on Figure 4.2.

The scenario shown is composed by four service functions. Each block can be seen as a separate execution unit, e.g. a virtual machine. Figure 4.2a shows that on Chaining-Box, each SF is contained in a CBox with all needed SFC functionality within it, and communication between SFs in the service plane is done directly, without a middle element. On the other hand, RFC 7665 (Figure 4.2b) uses extra levels of indirection, requiring that all communication goes through a Forwarder and possibly through and additional Proxy.



(a) Chaining-Box

(b) RFC 7665

Figure 4.2: Example scenarios with different SFC architectures.

As all the heavy lifting is now performed by the processing stages, there are only two tasks left for the underlying network: support packet classification and route packets between SFs.

Note that none of the stages discussed above implement classification, but rather expect that packets have already been classified and encapsulated with NSH before arriving to a CBox. This requires cooperation from a border router, either to classify packets when they enter the network, or to route them to a dedicated service function implementing classification that serves as an entry point for all ingress traffic. The first option breaks the decoupling offered by Chaining-Box, while the second can be achieved through simple routing rules on routers but may suffer from bottlenecks caused by the classifier SF. However, this can be mitigated by using well-known load balancing techniques.

Besides that, the network is also responsible for routing the packets between each service function. But this is done in an agnostic way, as network devices will route packets based only on the external transport encapsulation, be it VXLAN, Geneve, GRE or any other tunneling scheme, while the NSH will remain encapsulated in the tunnel.

4.2 Control Plane

The same attributions of the control plane from RFC 7665 also apply to Chaining-Box. A controller needs to setup rules on forwarding elements to specify the chains. In Chaining-Box terms, the Fwd stage of each managed CBox needs the entries in their forwarding tables to effectively enable the chaining. These actions are handled by the control plane, which is divided in two separate parts: an agent running on each CBox and a remote controller, as shown in Figure 4.3.

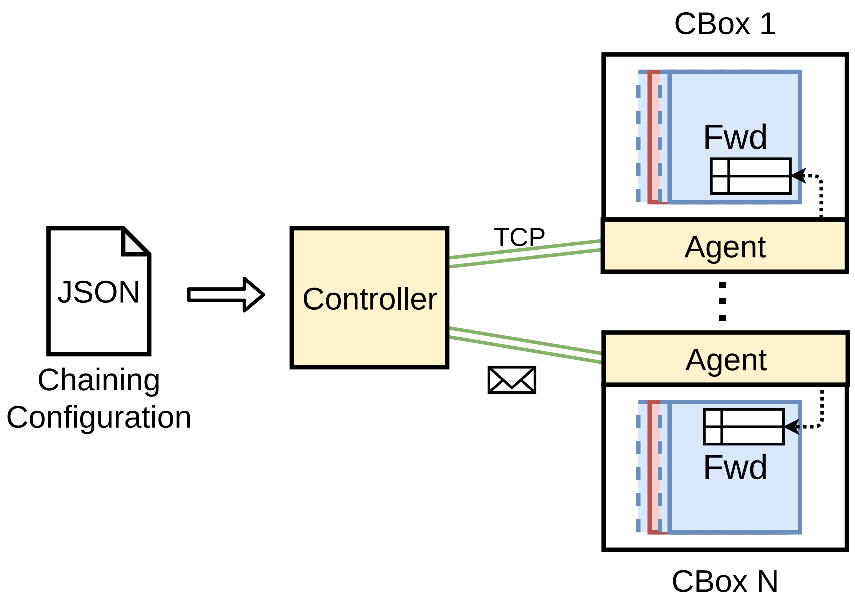


Figure 4.3: Chaining-Box control plane

Besides the processing stages discussed in Section 4.1, each CBox also has an agent that is responsible for all administrative tasks related to Chaining-Box on the execution unit. It handles communication with the controller and installs and configures the processing stages.

All agent-controller communication is based on a fairly simple southbound protocol, consisting on two types of messages: `Hello` and `Install`. As the names suggest, the first one is used by a CBox agent to notify the controller of its existence and its availability to participate in chains, while the second is used by the controller to send forwarding rules to each CBox instance, which are later installed by the agent on the Fwd stage. A third type of message to uninstall rules would also be required to fully support real-world deployments, but is left as future work.

The job of the Chaining-Box controller consists on:

1. **Parsing chaining configuration**: the composition of each chain is specified through a JSON file sent to the controller. It parses this file and generates the

forwarding rules which need to be installed on the CBoxes to realize the desired configuration. An example of such file is given in Section 5.6;

2. **Establishing connections with CBoxes**: on startup, every agent sends a `Hello` message to the controller using a TCP connection, which will be kept alive as long as the agent and the controller are running, so the controller knows which CBox are still functional;

3. **Installing rules on each CBox**: after a connection is established, if the controller has any chaining configuration addressing an specific CBox, its corresponding forwarding rules are sent through an `Install` message.

4.3 Architecture

The architecture description provided until now in this chapter was only a high-level view of the underlying principle guiding Chaining-Box's design. In practical terms, however, it may not be obvious how to realize the processing stages decoupled from both the network and from service functions.

Graph- or match-action-based packet processing as supported by VPP and OVS, respectively, could easily be used to implement the processing stages. These approaches would be fully transparent to functions, but would be coupled to the corresponding virtual switching technology used. A programmable-hardware-based solution also suffers from the same limitation, such as using SmartNICs or FPGAs to implement processing stages, since that would restrict the architecture to a few specific hardware devices.

An implementation at the service function level, using procedural programming, for example, to implement and enforce the order of processing stages and SF is also feasible, but requires changes to the function's source code, or implementing it entirely using specialized frameworks as some of the proposals discussed in Chapter 3.

Yet another option is to implement the stages as kernel modules, the alternative offered by Click-based solutions. This can be abstracted away by both the network and service functions, by acting completely in kernel space. Chaining-Box follows this same path, but instead of Click modules, it uses BPF programs, which are much simpler, have very little overhead, are native to the Linux kernel, have support from mainstream toolchains and can be transparently loaded and modified during runtime with little system disruption.

Figure 4.4 gives an overview of how Chaining-Box lays out physically in the system. The SFC mechanism is entirely placed between the SFs and the underlying network infrastructure, be it virtual or physical. By using BPF programs in the kernel, Chaining-Box is capable of handling different system models, be it a container, a virtual machine or a baremetal server. All the SFC administrative tasks are handled by the control plane and configured on the stages through the agent running in user space. Packets are processed on their way through the network stack requiring no cooperation from network devices.

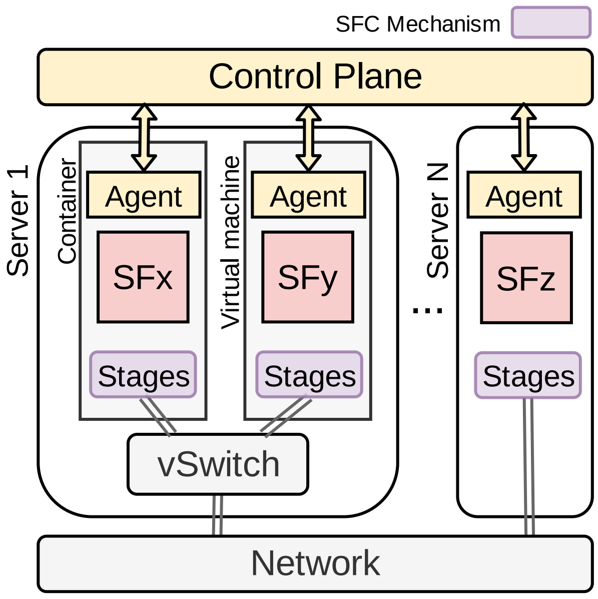


Figure 4.4: Chaining-Box physical layout

4.4 Comparison with other approaches

The separation of concerns between service and data planes provided by Chaining-Box becomes more evident if we put side-by-side the physical layout of different architectures such as in Figure 4.5. This figure compares Chaining-Box against two other architectures: a standard one where the SFC functionality is all provided by a virtual switch, e.g. OVS, and PhantomSFC. The boxes in purple highlight where the SFC mechanism is positioned in each architecture. All three examples implement the same chain, going through `SF1` and `SF2`, in that order.

Chaining-Box and the standard are very similar in terms of how many hops each packets needs until the end of the chain. The biggest difference lies on where SFC actions are being executed in the system. While Chaining-Box confines everything to each VM, the standard architecture has it all inside the virtual switch, coupling the chaining mechanism to the underlying infrastructure.

Although PhantomSFC also removes the SFC mechanism entirely from the network, this comes at a high cost in terms of performance, since the Service Function Forwarder (SFF) is fully virtualized and becomes the center of a star topology with the other SFs. This way packets have to go back and forth to the SFF before they finally exit the chain.

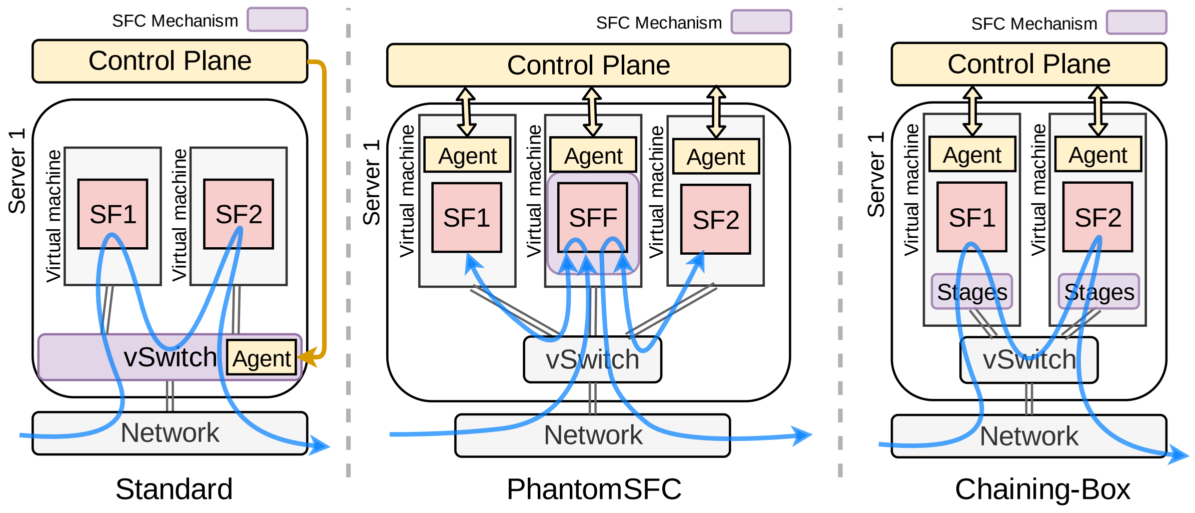


Figure 4.5: Comparison between Chaining-Box and other architectures

Chaining-Box is able to combine the benefits of both approaches while avoiding their pitfalls. The following chapters discuss in great detail how each component of the architecture is implemented, and the extra benefits added by the use of BPF technology in the dataplane.

Chapter 5

Prototype Implementation

To implement the processing stages discussed in the previous chapter in a fully transparent manner, Chaining-Box uses BPF programs inside the Linux kernel. The following sections discuss the encapsulation types used (§5.1), which BPF hooks were used and how each stage was implemented (§5.2), how Chaining-Box supports different application types (§5.3) and environments (§5.4, possible optimizations when using containers (§5.5), and how the control plane controller and agent were implemented (§5.6).

5.1 Additional headers

As stated before, Chaining-Box makes use of packet encapsulation to provide SFC. Packets traveling between service functions are encapsulated with both an external transport encapsulation as well as with the NSH. In the prototype built the transport encapsulation corresponds to an additional Ethernet header. For this reason all CBox used in the tests are required to be in the same L2 network. The decision to use this type of encapsulation aimed to simplify the prototype, without loss of generality. Other encapsulation schemes such as VXLAN or Geneve could also have been used, but are left as future work. Simple changes to Dec and Enc changes would be sufficient to handle different encapsulation protocols.

Currently the stages are only capable of handling MD-Type 1 NSH headers, which have four 4-Byte metadata fields. However, for the purposes of this text, these fields don't have any special meaning and were just filled with zeroes. Added to the extra Ethernet header, the NSH sums up to 38 extra bytes of overhead for each packet.

5.2 Stages and layers

The main BPF feature leveraged by Chaining-Box is the ability to process ingress and egress packets on their way through the network stack, without needing any cooperation or knowledge from the applications running in the system. Combined with the use of the NSH, this principle allows this implementation to be transparent to both the network and service functions. The following subsections describe the inner workings of each stage.

5.2.1 Decapsulation (Dec)

Since the stages perform header-changing operations on the packets, ideally they should be executed as close to the NIC as possible, to have complete power over ingress packets seen by the kernel and egress ones seen by the network. On the RX side, this lowest point is represented by the XDP layer, which in its native form is implemented directly by the device driver. At this point, the OS has neither parsed incoming packet metadata nor allocated socket buffers for it, allowing XDP programs to make arbitrary changes to the packets without the OS' knowledge. For these reasons, this was the layer chosen for the Dec stage, as shown in Figure 5.1.

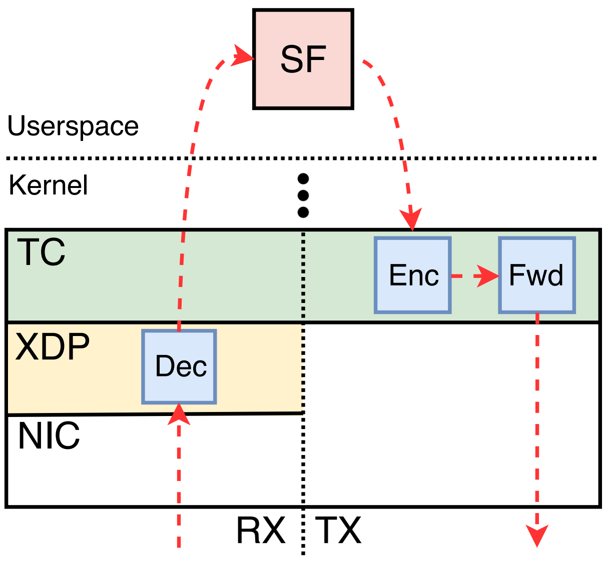


Figure 5.1: Stages implemented as BPF programs inside the Linux kernel

The Dec stage is implemented in less than 100 lines of C code and can be summarized to the following actions, illustrated by the flow diagram of Figure 5.2: (1) detect if this is the destination of the current Ethernet frame; (2) check if the packet received contains an NSH header; (3) if so, parse it's IP 5-tuple; (4) use it as key to save the NSH to a temporary table implemented as a BPF hash map shared with the Enc stage; and (5) remove the transport and NSH encapsulations. After all these steps are complete, the packet is sent up the kernel stack for further processing. Packets that do not contain the expected transport and NSH encapsulations are just ignored and passed along, so the stage does not interfere with other programs and services running on the same machine.

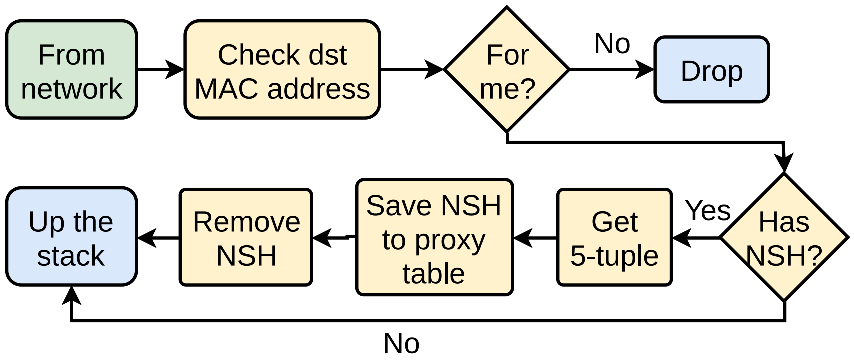


Figure 5.2: Flow diagram of Dec stage

When saving NSH information to the map (proxy table) the key associated with this value is the internal packet's 5-tuple, ignoring the outer encapsulation, as shown in Figure 5.3. This same key is used later by the Enc stage to match outgoing packets for re-insertion of the NSH. Of course, this approach has limited support for SFs that change the packet's 5-tuple such as NAT or VPN endpoints. More complex and clever matching algorithms can be implemented as well, requiring just implementing new Dec and Enc stages, which can be loaded and replaced atomically by the BPF system.

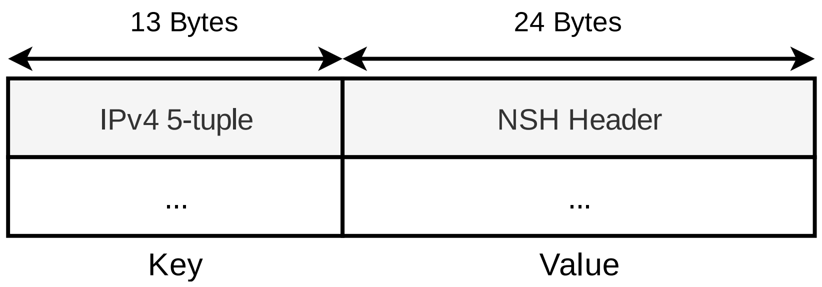


Figure 5.3: Proxy table shared between Dec and Enc stages.

5.2.2 Encapsulation (Enc)

The initial envisioned use case for the XDP layer was to provide a faster way to make early decision about incoming packets to improve the performance of firewall and DDoS mitigation services, for example. For this reason, this layer is only present on RX, and not on TX, rendering it unable to implement Enc and Fwd stages. Although there seems to be plans by the community to extend the XDP to egress as well, it has not been a priority at the moment, needing more compelling use cases and killer applications to steer its development. Thus the last two stages are implemented on TC, which is the lowest layer on TX side. The context seen by programs in this layer is also a Layer-2 packet, allowing fairly generic changes to the packet as well.

The Enc stage receives the packet after it has been processed by the service function. It's actions are summarized by the flow diagram in Figure 5.4. It prepares the packet for the Fwd stage by (1) parsing its 5-tuple; (2) using that as a key to the BPF hash map shared with the Dec stage to retrieve the previously removed NSH header; (3) checking whether this was the last hop in the chain; (4) re-inserting the NSH and transport encapsulations on the packet; (5) decrementing the SI value from the NSH; (6) sending it to the next stage. In case the pre-lookup of the forward table in step (3) shows that this is the last node in the chain the NSH re-insertion step is skipped and the packet is sent directly to the network. The check consists on checking the flags field stored alongside the next hop address. If this field is set to 1, that indicates the end of the chain.

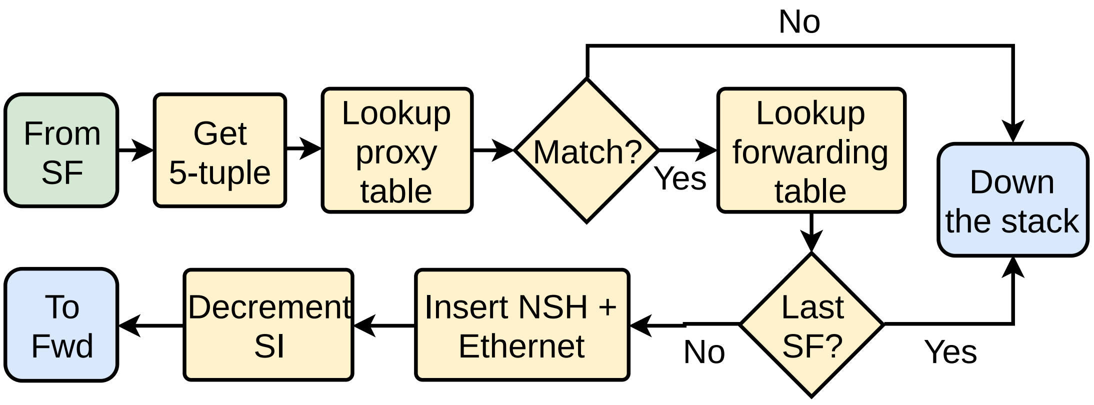


Figure 5.4: Flow diagram of Enc stage.

The TC layer has support for internal chains of filters and actions (not to be confused with SF chains), that are executed in order defined by a priority value associated with each one. Enc and Fwd stages are implemented as filters in the same chain, with Enc having a higher priority than Fwd, and thus executing first. Similar to the Dec stage, packets that do not match a 5-tuple stored in the table are deemed unrelated to the SFC chaining, and are just passed along the stack normally, causing no interference with other services.

If the packet is successfully re-encapsulated, Enc returns a `TC_ACT_UNSPEC` value, instructing TC to execute the next filter in its internal chain, which is configured by Chaining-Box to be the Fwd stage. This setup, however, suffers performance penalties as these two disjoint programs will have to perform packet header parsing separately. This design was chosen to decouple both stages and make it easier to replace each one separately as needed, in order to provide different functionality, for example. However, combining both stages into one is a simple matter of writing a BPF program that does the job of both.

5.2.3 Forwarding (Fwd)

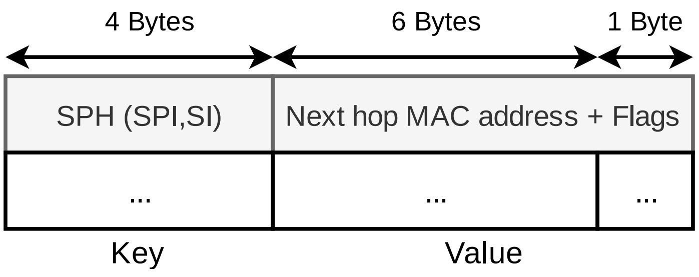


Figure 5.5: Lookup table used by Fwd stage to get next hop information.

Finally, the Fwd stage receives packets already encapsulated with the NSH and uses SPI and SI values from it to lookup the internal forwarding table (Figure 5.5), populated by the control plane agent. This table is also implemented as a BPF hash map and uses the SPH (SPI + SI) as the key to retrieve the address of the next hop. This address is then written to the packet's transport encapsulation and it is sent back to the network to be forwarded to the next SF in the chain. All these actions are summarized by the flow diagram in Figure 5.6. Note that packets not containing a corresponding entry in the forwarding table are dropped, this situation indicates bogus NSH-encapsulated packets received by the Dec stage or a misconfigured Fwd stage. In either case, this packet should not go back to the network, so it is dropped.

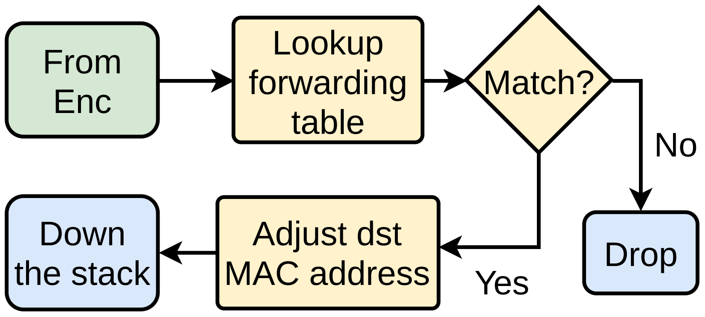


Figure 5.6: Flow diagram of Fwd stage.

5.3 Application support

The careful reader may have noticed the discussion of the SF stage was skipped in the last section. This was intentional as it is not directly implemented by Chaining-Box itself, but rather by other existing service function applications. The architecture supports two kinds of applications discussed in Section 2.2: kernel-supported and kernel-only. Figure 5.7 illustrates how the stages interact with each application type.

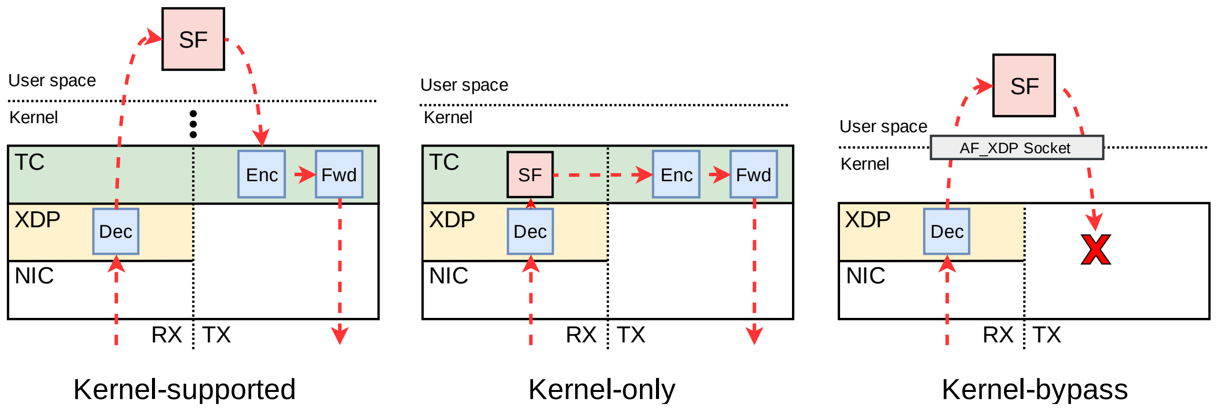


Figure 5.7: Chaining-Box supports different kinds of service functions

Kernel-supported applications reside entirely in user space, and represent the simplest case for Chaining-Box. As the stages are implemented as BPF programs loaded to the kernel, all operations necessary to steer packets through chains is done without any interaction from the application, making it totally agnostic to the SFC environment. Packets are processed and modified by the stages on their way through the network stack.

Kernel-only functions, on the other hand, can be other BPF programs also loaded to the kernel, or implemented by *iptables* rules or kernel modules. For KO apps located above the TC layer, there is little difference from the KS case, as Chaining-Box stages will behave as an middle-man to the network and no special treatment is needed by the architecture. The same is true for functions implemented on the ingress TC direction, as illustrated by 5.7 in the Kernel-only case. On TC-egress, however, a small change on how

Type	Dec?	Enc?	Fwd?
Kernel-supported	Yes	Yes	Yes
Kernel-only	Partial	Partial	Partial
Kernel-bypass	Yes	No	No

Table 5.1: Summary of the Chaining-Box stages supported for each SF implementation type.

the stages are attached is necessary. The SF must be placed in the same TC chain as Enc and Fwd stages, but with a higher priority, so it is executed first. Also, it would require a small change to it's source code to instruct TC to keep executing the next filters, i.e. it should return `TC_ACT_UNSPEC` after successfully processing a packet.

SFs implemented as XDP programs represent a tricky case not fully supported by Chaining-Box at this moment. Since XDP does not have an internal chaining mechanism like TC, it is currently not possible to load multiple XDP programs to the same interface simultaneously. Although there has been recent proposals for such mechanism[1], it is still under development and it is not certain whether it will ever reach the upstream kernel. One possible workaround is to execute the XDP application as a BPF tail call after Dec stage execution, but has not been tested yet with Chaining-Box.

Lastly, kernel-bypass functions are not supported by Chaining-Box. Since these applications completely bypass the kernel stack, the processing stages cannot be executed. `AF_XDP` sockets represent a partial solution for this problem, supporting the execution of XDP programs prior to sending packets directly to user space, bypassing upper layers in the networking stack. Application built with DPDK, for example, could use the provided `AF_XDP` poll mode driver (PMD) to use this socket family, which would permit the execution of the Dec stage. But since XDP is present only on RX, the egress stages Enc and Fwd are not contemplated by this solution. Table 5.1 summarizes the support offered by Chaining-Box to each application type.

5.4 Environment support

In this section, the word 'environment' refers to the overall operating system environment where the service function is being executed, which can be: (1) baremetal; (2) virtual machine; or (3) container.

When it comes to how Chaining-Box is laid out in the system, the first two environments are the same, both have a full isolated kernel that can host the BPF processing

[1]https://www.spinics.net/lists/netdev/msg602065.html

stages. Although containers share the same kernel with the host and other siblings, each has its own network interfaces, usually one end of a veth pair or even a passthrough physical interface. Since BPF programs on both XDP and TC are attached to an specific interface, these programs are isolated from one another. From Chaining-Box's point-of-view, they behave the same as if each container had a dedicated kernel stack, enabling the architecture to work without requiring any changes.

However, there is a possible optimization for the container case. If consecutive containers in a chain belong to the same host, they can be interconnected directly through a veth pair, called a direct link, instead of having to go through a virtual switch. This brings three main performance benefits: (1) veth pairs can achieve higher throughput rates than going through a virtual switch; (2) NSH and transport encapsulations are not necessary, as this is a direct link; and (3) as a consequence of the lack of encapsulation, stages Enc and Dec from the source and destination containers, respectively, can be bypassed. More details about this optimization are discussed in the next section, while the overall performance improvement observed is demonstrated in Section 6.7.

5.5 Containers and direct links

Figure 5.8 better illustrates how Chaining-Box can use direct links. On that scenario, `SF1`, `SF2` and `SF3` are part of the same chain and are executed in sequence. Each is hosted by a separate container, and all three containers belong to the same host. In this case, Chaining-Box can instruct the agent to use veth pairs between each consecutive SF to speedup packet steering through the chain. Fox example, the CBox hosted in Container 1 can have its Enc and Fwd stages disabled, and forward packets directly through the veth pair to the next hop in the chain. `SF2` also has its Dec stage disabled, since no encapsulation was used by `SF1`. Since `SF2` and `SF3` are also connected through a veth pair, Enc and Fwd from `SF2` and Dec from `SF3` are also disabled. In this example, through the use of 2 veth pairs, 5 stages were omitted, improving the end-to-end chain latency as will be shown later in this text.

This optimization, however, introduces two corner cases that must be handled by the controller to keep the chaining functioning correctly. They happen when the Dec stage is omitted but Enc still has to be run. The first case is when two CBoxes that are directly linked are followed by a another without direct link (Figures 5.9a). In this case, the second node will have its Dec stage omitted because of the direct link connecting it to the previous function, but will still have the Enc stage as it is needed to communicate with the following node. The second case happen when when two directly-linked nodes terminate

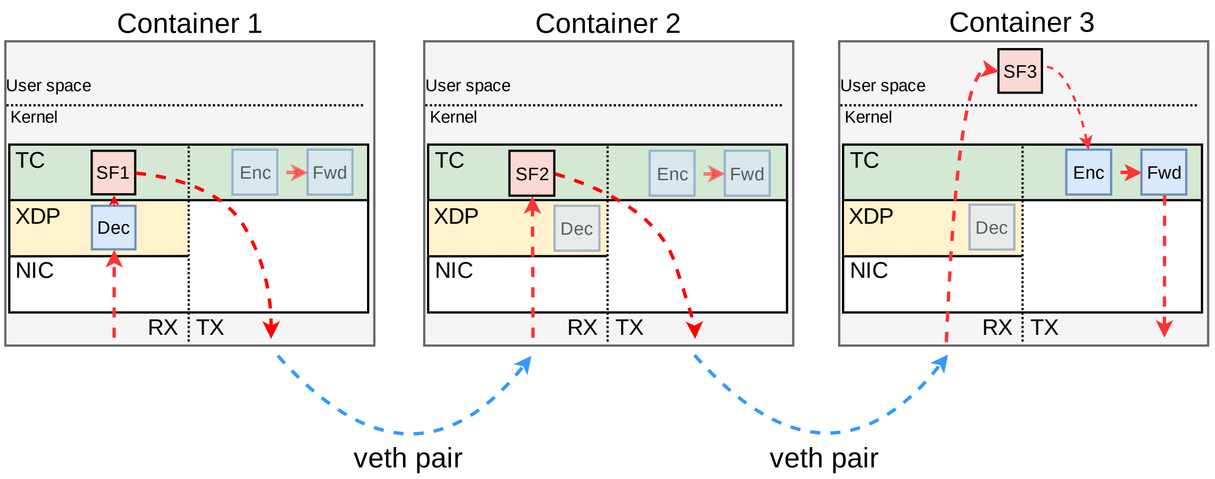


Figure 5.8: Example of use of veth pairs to simplify packet steering between SFs in containers in the same host

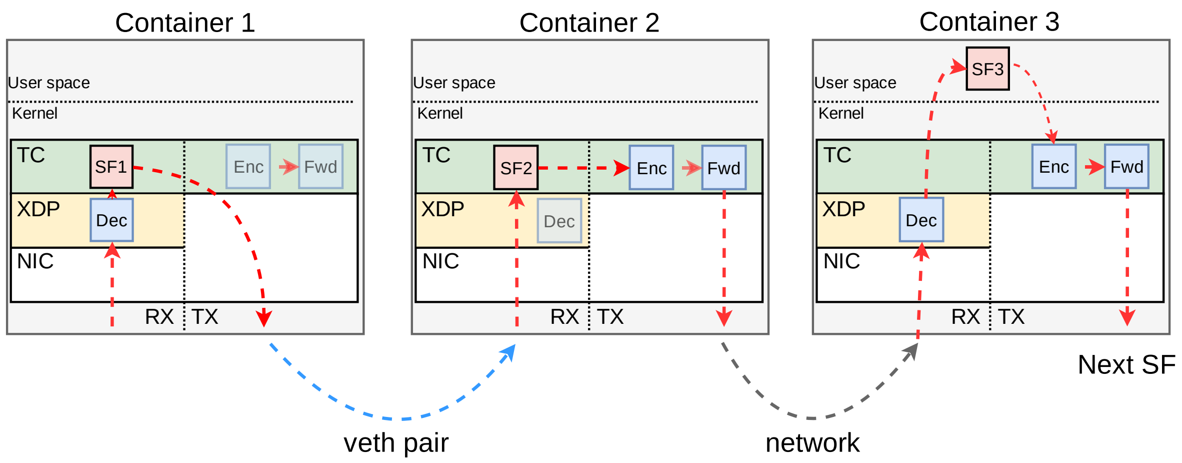
a chain (5.9b). Since the last element in the chain will always have to decapsulate the packet, it's Enc stage cannot me omitted, but will omit Dec if packets come from a direct link. In such cases, Enc will lookup the table for the corresponding NSH header for the packet but no entries will be found, since no packets were decapsulated and Dec was never executed.

However, these situations can be detected early by the controller when configuring the chains, since it knows chain layouts and which nodes will be configured with direct links. When one the corner cases are detected, the controller simply adds artificial entries to the proxy table to Enc stage of the faulty node, as if they were generated by a packet decapsulation on Dec. This has the obvious limitation of not permitting that NSH metadata be carried through the chain, since the controller is unable to predict the contents of each metadata header beforehand.

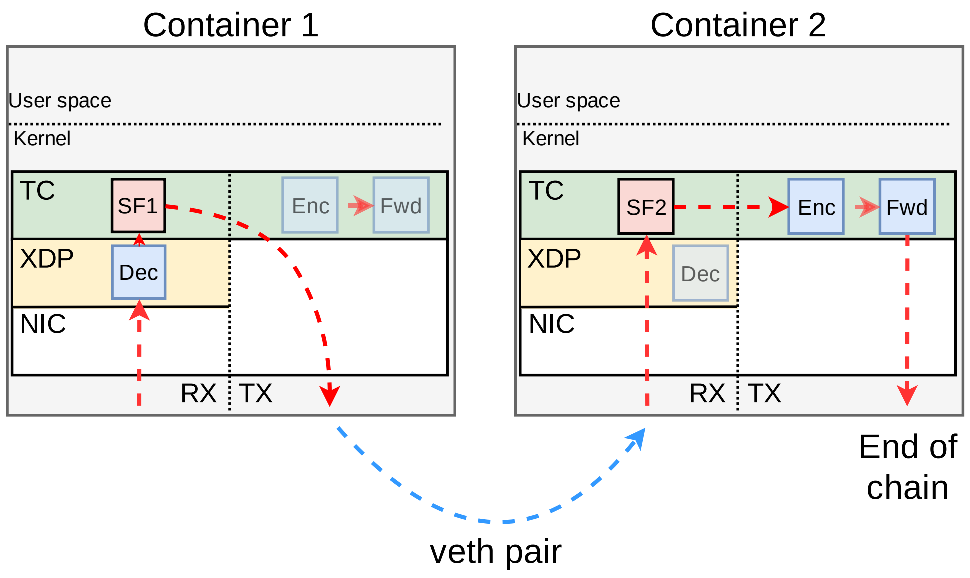
5.6 Agent and controller implementation

Both the controller and agent are implemented in Golang. All code to interact with the BPF system is written in C using *libbpf* and *iproute2*, and called from within the Golang agent. The agents communicate with the remote controller using TCP connections, that are kept alive as long as the agent is running. The controller keeps an internal table of all connected agents, which is used during chain configuration.

A system administrator can specify the chains to be used through a simple JSON file. This file is parsed by the controller and the generated rules are sent to the cor-



(a) Case 1: Two directly linked functions followed by a remote node



(b) Case 2: Two local nodes, directly linked ending a chain

Figure 5.9: Example of corner cases of direct link optimization that require special treatment by the controller.

responding agents through their TCP connections to be installed by the agent on each CBox. Figure 5.10 shows and example of JSON file to configure an environment with two chains. Each chain is configured with 2 functions each. Each function must have a tag by which it will be identified in when specifying the chains, besides the host in which it will be deployed. The value `type` was used to indicate which of the implemented test functions the controller should deploy, but should be changed to a more generic way to specify any kind of function.

For this input file, the controller will generate one forwarding rule for each of `sf1` and `sf2`, and two for `sf3`, as shown on the right side of Figure 5.10. The MAC address of the input interface of each SF is sent to the controller when the agent sends a Hello message, and these values are later used to configure the forwarding tables accordingly. Since both chains end in `sf3`, its entries have the MAC address zeroed out and the flags

```
{
  "chains": [
    {
      "id": 100,
      "nodes": ["sf2","sf3"]
    },
    {
      "id": 200,
      "nodes": ["sf1","sf3"]
    }
  ],
  "functions": [
    {
      "tag": "sf1",
      "type": "tc-redirect",
      "host": "server1"
    },
    {
      "tag": "sf2",
      "type": "tc-redirect",
      "host": "server1"
    },
    {
      "tag": "sf3",
      "type": "tc-redirect",
      "host": "server2"
    }
  ]
}
```

SF1 forwarding table

SPH (SPI,SI)	Next hop MAC address + Flags	
(200,254)	SF3 MAC	0

SF2 forwarding table

SPH (SPI,SI)	Next hop MAC address + Flags	
(100,254)	SF3 MAC	0

SF3 forwarding table

SPH (SPI,SI)	Next hop MAC address + Flags	
(100,253)	ZERO	1
(200,253)	ZERO	1

Figure 5.10: Example of JSON configuration file given to controller and the corresponding rules generated.

field is set to 1, indicating end of the chain.

It is also worth noting that the control plane automatically determines when direct links can be used in chains. This information can be derived from the `host` fields from consecutive functions in a chain. If two SFs are hosted by the same server and are executed in sequence in a chain, the control plane automatically connects them with a direct link. On a full-fledged NFV environment, the JSON configuration file would represent the output of a placement algorithm, while the controller described here would carry on the execution and deployment of such placement.

Lastly, the choice of JSON as the format to define the chaining configuration was due to its simplicity and builtin support in many languages, such as Golang. An additional benefit is that files can be easily validated using JSONSchema[2]. The support for other NFV-specific formats such as TOSCA [Garay et al., 2016] is desirable for better integration with existing NFV frameworks but is left as future work.

[2]https://json-schema.org/

Chapter 6

Experimental Evaluation

The performance of the prototype was evaluated considering different aspects, which are summarized in the following sections. Besides the physical setup used (§6.1), they present some implementation statistics (§6.2) and the overhead imposed by each stage (§6.3). After that, the methodology used in the tests is presented (§6.4), followed by an analisys of the architecture's barebones performance in the absence of functions considering two different variables: packet size (§6.5), chain length (§6.6). The performance benefits inherited from using direct links is also shown (§6.7) and a comparison with an emulated architecture is presented (§6.8).

6.1 Environment Setup

The logical setup used during tests consisted on two hosts communicating through an SFC-enabled environment composed of multiple functions in a chain, as shown by Figure 6.1. The chaining was applied in a single direction, even though host reachability went both ways.

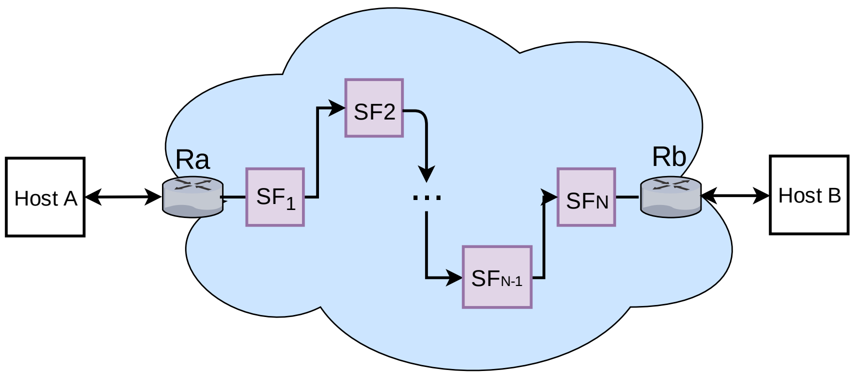


Figure 6.1: Logical setup used for experiments.

The physical setup consisted of a single server with an Intel Core i7-7740X CPU

@ 4.30GHz, 16 GB of RAM and two network cards, an Intel X710 and a Netronome Agilio CX 4400 with two 10 Gbps interfaces each. Each service function was deployed to a separate Docker container, all connected by an Open vSwitch (OVS) bridge. The NICs were connected back to back, one to be used by the packet generator and the other to act as the device under test. Two different approaches to packet generation were used: *pktgen-DPDK* [Wiles, 2016] for throughput tests (Figure 6.2) and two separate namespaces with an interface each (Figure 6.3) for latency measurements. In both cases, one interface from the generator was used for RX and the other for TX.

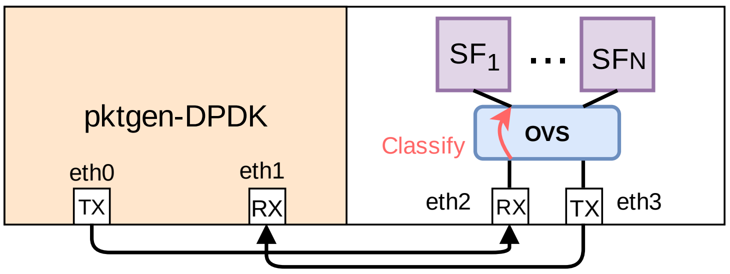


Figure 6.2: Physical setup used for throughput experiments.

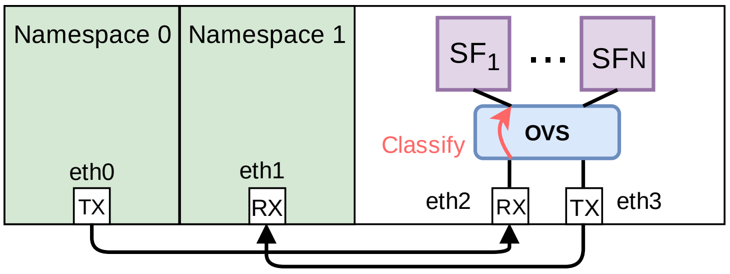


Figure 6.3: Physical setup used for latency experiments.

The OVS bridge was configured with two rules: one to provide packet classification and insertion of the NSH header and another to forward packets as a common L2 learning switch. Note that classification could be done in different ways, like generating pre-classified packets or routing all traffic to a single entry-point container to act as a classifier. We chose to use OVS because of its built-in support for NSH in recent versions and better performance compared to the other viable options at the time of testing. All packets arriving from the traffic generator were matched against the classification rule and if they matched the pre-defined flow, an NSH header would be added to the packet and it was re-circulated in the switch to match the L2 forwarding rule and be forwarded to the first SF in the chain based on the outer transport encapsulation, an Ethernet header in this case.

All tests used a single type of service function: a KO function implemented as a BPF program on the TC hook that received packets from one interface and redirected to a second one, named `tc-redirect`. This was chosen as the simplest SF possible, allowing to measure the architecture's performance, without having to account for extra overheads imposed by the service function. Nonetheless, this SF's overhead proved to be considerably higher the stages', as is shown later in this chapter.

6.2 Program statistics

Table 6.1 shows the actual size of each BPF program used to implement the stages. All programs were implemented in less than 400 lines of C source code, counting common code shared by the stages and excluding header files. After being compiled and loaded to the kernel, the biggest program (Enc stage) was 2072 Bytes in size. Even with the extra locked memory used by the kernel to allocate the data structures to hold the program each occupied only 4096, summing up to a mere total of 12 kB for all stages. The amount of memory per program is constant and dependent on the bytecode generated by the compiler and jited code generated by the kernel.

	LoC	BPF insns	Bytecode (B)	Jited (B)	Locked (B)
Dec	78	117	992	637	4096
Enc	78	232	2072	1197	4096
Fwd	40	54	584	359	4096

Table 6.1: BPF program statistics

	Entries	Key (B)	Value (B)	Data (B)	Locked (kB)
Source MAC	2	4	6	20	4
Proxy table	2048	13	24	75776	212
Forward table	2048	4	7	22518	164

Table 6.2: Memory footprint of tables

Besides program instructions, the kernel also needs to allocate space for the maps used by them. Table 6.2 shows the memory footprint of the maps used by all programs. The amount of memory needed is a function of the sizes of keys and values and the amount of entries on each table. This number is hardcoded during map declarations on the source code, but can easily be changed, requiring recompiling and reloading the stages. The real number of bytes occupied in memory is represented by the locked memory reserved by the kernel, shown in the last column. Still, the total memory required for map storage on each node during tests was around 350 kB. Even if the proxy and forwarding tables were configured with 1 million entries each, the total memory required by all maps would be less than 200 MB, a modest requirement for today's hardware.

These numbers demonstrate that the memory footprint of Chaining-Box stages is fairly minimal, which possibly allows for the deployment of many instances on the same server with little problem.

6.3 Processing overhead

An important metric to understand the Chaining-Box's performance is the processing overhead added by each program to each packet, i.e. how much time each stage takes to execute. For such, a single chain with 2 SFs was used. The length is important because the last element in a chain will have lower overhead since the packet does not have to be re-encapsulated and the Fwd stage is skipped entirely.

Each stage was instrumented to collect timestamps at the beginning and end of its execution using a native BPF helper, which introduce approximately 20 ns of extra delay. The average was taken over 1000 executions, and the values measured subtracted of the extra overhead described are shown in Table 6.3.

Thus the increase in latency introduced by a single CBox is around 4.6 μs in the worst case in which all stages need to be executed, e.g. for legacy functions. For an NSH-aware function, for example, in both Enc and Dec stages can be omitted, the extra latency becomes only the one introduced by the Fwd stage, around 540 ns.

Compared to the time spent to redirect the packets on the SF used, the overhead added by the stages is minimal. The CPU usage of each BPF program was measured using `perf` when generating packets at 10 Gbps with `pktgen`. The percentage values shown in Table 6.4 are related to the time the system spent running only BPF programs, including the SF (listed below as `tc-redirect`). The remaining 0.5% corresponds to administrative tasks executed by the kernel to run the programs. Overall, BPF program execution corresponded to 16% of CPU load during the test, which also had processes for OVS and `pktgen` running in parallel on the same server.

The high overhead imposed by a simple function such as `tc-redirect` can be explained by its use of the `bpf_redirect` helper, which redirects packets to another

	Avg. Latency (ns)
Dec	2208.55
Enc	1840.60
Enc (end of chain)	1197.08
Fwd	540.66

Table 6.3: Additional processing time introduced by each program.

	BPF CPU time (%)
Dec	0.23
Enc	0.25
Fwd	0.09
tc-redirect	98.93

Table 6.4: CPU usage of each program during stress test.

output interface and is an expensive operation. An optimized version for this function called `bpf_redirect_map` tries to reduce the its cost by performing operations in batches, providing superior performance than the previous alternative, but is only available in the XDP layer.

Together with the memory requirements shown in the previous section, the very low CPU overhead imposed by the processing stages indicates that Chaining-Box is scalable, potentially allowing many CBox to be deployed on the same physical server.

6.4 Test Methodology

The subsequent sections present summarized results of several measurements on the prototype discussed in the previous chapter. All performance tests were based on two important metrics for networked systems: throughput and latency. The two different setups used for each type of measurement were discussed in the previous section.

The throughput values reported on all graphs were calculated as such:

$$Throughput = \frac{\text{packets received}}{\text{packets sent}} \cdot T_{gen} \tag{6.1}$$

The inputs to the formula were the results of sending packets uninterruptedly for 10 seconds at a generated rate (T_{gen}) of 10 Gbps. In addition, all SFs used in this type of tests consisted of the default `tc-redirect`.

The maximum packet size in all tests was restricted to 1462, which corresponds to the default MTU minus the overhead of the encapsulation used by the Chaining-Box prototype. When not stated otherwise, this was the size used to generate packets during the tests.

Latency tests used `ping` to generate and measure latency in terms of the round trip time (RTT) of packets. As only the traffic coming out of the traffic generator was classified by OVS, changes to the RTT for different chain configurations represented the actual extra latency added by Chaining-Box.

All latency results are a median of 120 repetitions with the top 10 and bottom 10 outliers removed, totaling 100 samples. When present, error bars represent standard deviation.

Due to resource limitations, the entire test setup was reduced to a single server, and had to rely on default OVS for packet switching. This configuration has its own performance issues, not being capable of reaching peak performance for every packet size at 10 Gbps, as shown in the next section. Our baseline consists of communicating between the two endpoints without any chaining, only forwarding by OVS. The default baseline latency measurement was around 0.5 ms.

6.5 Packet size

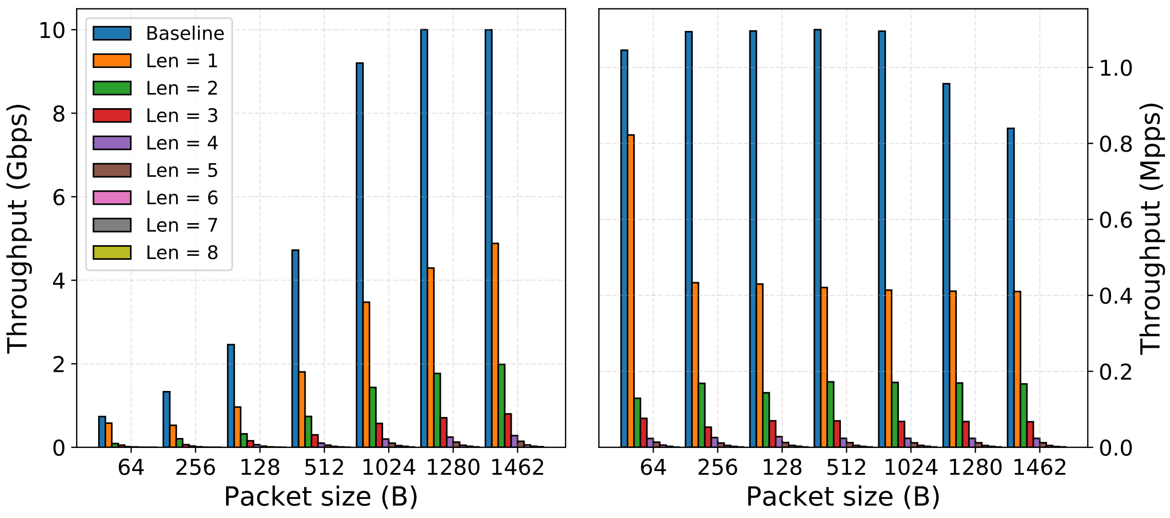
At high throughput rates, smaller packets tend to represent higher stress for networking systems as the number of packets to handle increases substantially, so packet drops are common. The impact of the packet size on both the throughput and latency of chains provided by Chaining-Box was measured and is shown on Figure 6.4.

For both throughput and latency measurements, the tests were repeated for chains without direct links and of varying length, from 1 up to 10, besides the baseline. Note that the throughput in Gbps decreases substantially as the packet size gets smaller (Figure 6.4a - left). This can be explained by the fact that the underlying OVS bridge is not able to handle the high number of packets being forwarded through it. Also for a given packet size, the throughput also decreases as the chain length grows, which is expected since each packet has to go back and forth between the SFs and the bridge more times for a longer chain, increasing the effective number of input packets on the bridge.

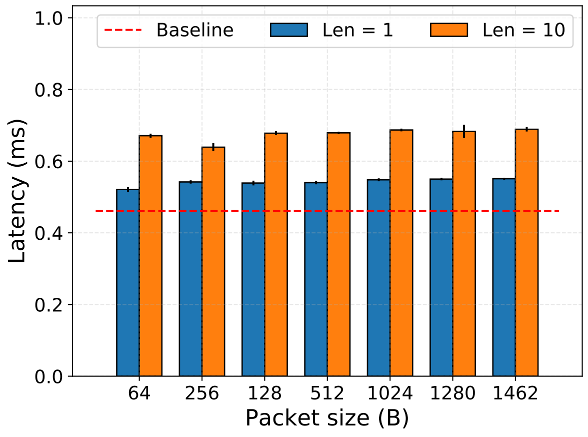
This explanation is backed by the same results plotted using a different unit, packets per second (Figure 6.4a - right). Except for 64 B packets, the rate in Mpps is stable for each chain length even with different packet sizes, suggesting that the actual limit imposed by the underlying infrastructure is indeed bound to the number of packets, instead of the actual number of bytes being forwarded. Since everything is being run on the same server, many events are triggered once a packet is received from the packet generator. Interruptions generated by the NIC, rule processing by OVS, BPF program execution for each stage and the service function are just some examples. These added to the poor performance of the service function used accounts for the performance degradation observed.

The outlier seen for 64-Byte packets can only be explained as a fluctuation on process scheduling and resource contention that favored the results at the time of the test. This hypothesis is derived from the fact that many other executions of the same test did not yield this outlier.

Another important information derived from these results is the low throughput of our baseline for small packets, indicated by the blue bars in Figure 6.4a. As noted previously, our baseline already suffered from limitations of the test setup used during experiments. Thus, part of the lack of performance observed on all tests is not purely due to Chaining-Box's mechanism, but mainly to the underlying test environment.



(a) Throughput in Gbps (left) and Mpps (right)



(b) Latency

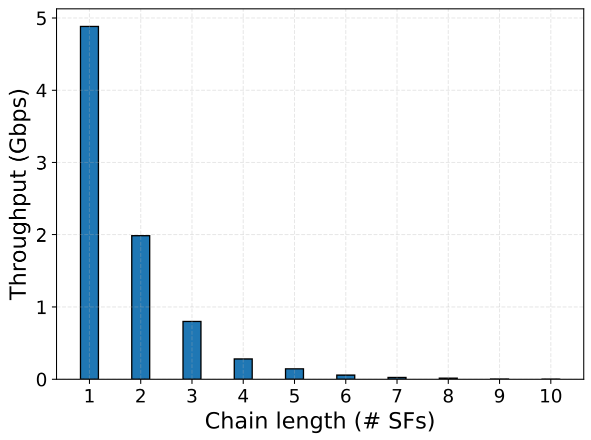
Figure 6.4: Measurements of the impact of packet size on Chaining-Box performance

In terms of latency, the impact of the packet size is minimum, as shown in Figure 6.4b. Only the results for chain lengths 1 (minimum) and 10 (maximum used in tests) are shown, however the trend remains the same for all other lengths. This plateau is due to the fact that the stages only operate on the packet headers, and the SF used in the tests does not modify the payload either. Thus, the packet size has little to no effect over latency at the accuracy used during measurements. Of course, the longer packets require extra processing time to be transmited by the NIC and OVS, but this difference is small compared to the fluctuations caused by process scheduling, context switching, and other operating system events that happen simultaneously, which end up masking out that extra delay.

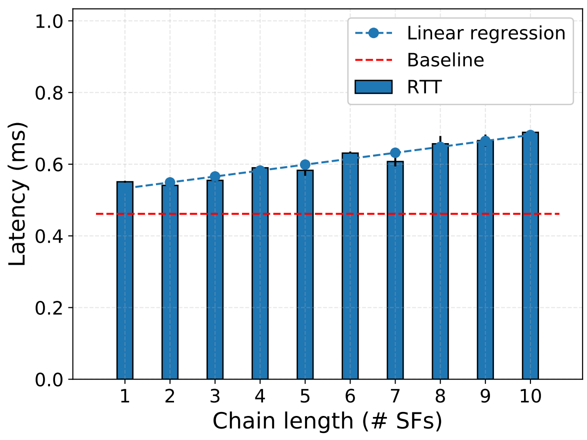
6.6 Chain length

The impact of the length of the chain on the throughput is similar to what was discussed in the previous section regarding the packet size. As the chain gets bigger, the underlying switch has to process more and more packets, which overloads the system and causes a acute performance degradation, as shown in Figure 6.5a.

In terms of latency, Figure 6.5b shows that latency grows linearly with each additional SF. A linear regression model of the measurements, also show in the graph, has a derivative of 16.5 μs, which represents the extra delay incurred by each extra SF in the chain, out of which only 4.6 μs are due to Chaining-Box processing stages as shown in Section 6.3. The remaining portion of this delay can be attributed to OVS' match-action algorithm and packet transport in and out of the system.



(a) Throughput

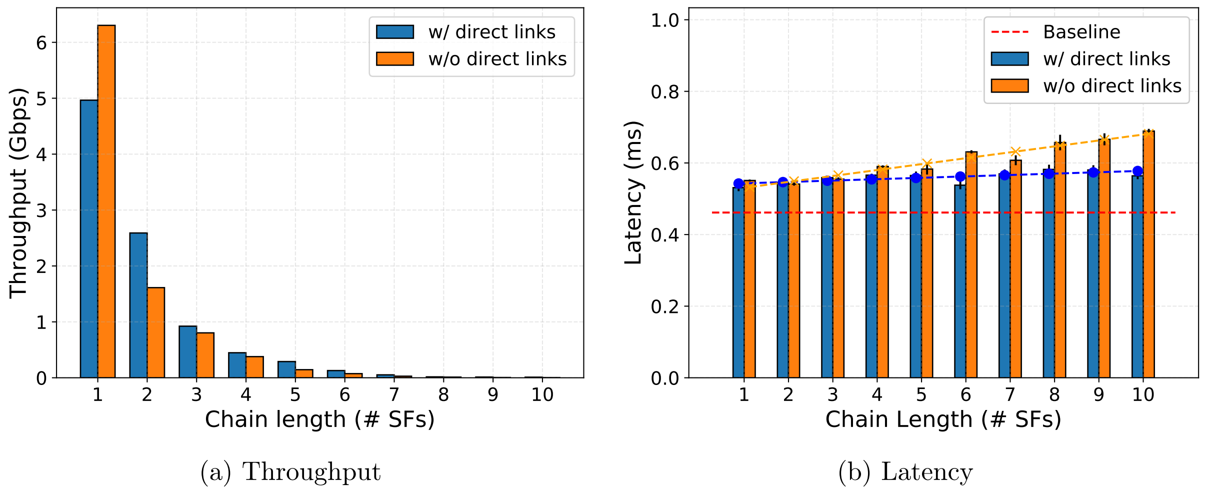


(b) Latency

Figure 6.5: Measurements of the impact of chain length on Chaining-Box performance

6.7 Benefit of Direct Links

Figures 6.6a and 6.6b show the performance results demonstrating the actual benefits of using direct links to connect local containers as dicussed in Section 5.4. With direct links, some Chaining-Box stages and OVS can be avoided altogether as the containers are directly connected by veth pairs. This reflects into the observed end to end throughput which is higher for the case with direct links and also into the latency, which is lower in that case. While the latency without direct links increases by 16.5 μs for each new SF, with direct links this increment comes down to 3.9 μs, or approximately 23.6% of the former.



(a) Throughput (b) Latency

Figure 6.6: Evaluation of improvement provided by use of direct links.

6.8 Comparison with standard architecture

As stated before, few other SFC architectures aim to decouple service and data planes as Chaining-Box, PhantomSFC [Castanho et al., 2018] being one of the exceptions. For this reason, we try to make a qualitative comparison between both by emulating PhantomSFC's behavior of relying on a centralized (but decoupled) forwarder by creating a chain that forces packets to be processed by the same SF at every step in the chain, as illustrated in Figure 6.7 for a chain of length 2.

Having a centralized forwarder is similar to having one SF that is visited at the beginning, end and at every hop of a chain, clearly, this forces packets to visit more SFs

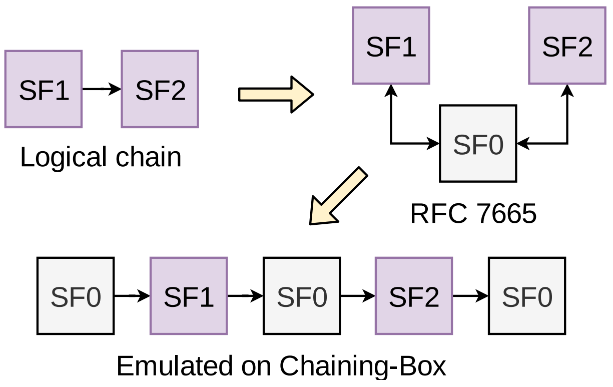


Figure 6.7: Example of how a PhantomSFC-like architecture is emulated with Chaining-Box for experiment the experiment.

before exiting the chain if compared to chains implemented with Chaining-Box, in which functions are able to communicate directly. For a chain of length N, a corresponding chain in the emulated architecture has $2N + 1$ nodes.

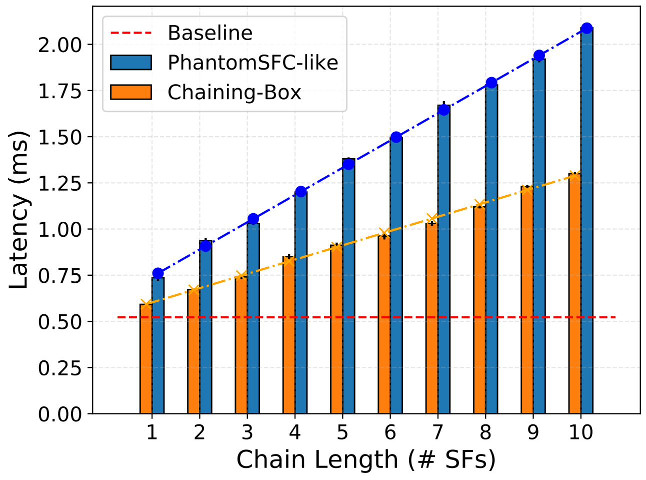


Figure 6.8: Latency comparison between Chaining-Box and an emulated PhantomSFC-like architecture.

Since all experiments were done locally, the latency values measured all fall in the range of hundreds of microseconds, which is too subject to interference from fluctuations caused by the OS, which can mask the actual latency variations that we are trying to measure. For this reason, this test used a variation of the `tc-redirect` SF discussed before, which included an extra processing delay simulating the time an actual SF would spend performing some operation on the packet, named `tc-redirect-delay`. The delay consisted on an empty `for` loop with 10000 iterations.

Clearly Chaining-Box will yield better performance, since longer chains incur performance degradation as demonstrated previously on Section 6.6. Figure 6.8 shows a comparison of the latency observed for packets traversing chains of growing lengths in the emulated scenario and standard Chaining-Box chains. Latency increases at much higher rate for the emulated chain than for Chaining-Box, an example of how the proposed architecture benefits from direct communication between functions.

Chapter 7

Discussion

In face of the architectural design and the experimental evaluation discussed before, this chapter discusses some of the advantages (§7.1) and disadvantages (§7.2 of Chaining-Box compared to other architectures. Afterwards, challenges and limitations of current technology faced while developing Chaining-Box are presented (§7.3).

7.1 Advantages

The advantages of Chaining-Box can be split in two categories: the ones derived directly from the architectural concept proposed (§7.1.1) and others that stem from the choice of using BPF as the technology to implement the processing stages (§7.1.2). These are discussed in the following sections.

7.1.1 Benefits offered by the architecture

Transparency to both SFs and underlay network: by moving all SFC functionality to the SF's executing unit, the need for NSH support by the network infrastructure and the service function are lifted. In case the function is NSH-aware, this transparency is not needed, so both Enc and Dec stages can be disabled, improving function performance. However, the facilities offered by Chaining-Box's processing stages allow legacy functions, i.e. NSH-unaware, to be part of a chain without any changes to its source code.

Flexible support for different proxy operations: in the architecture proposed by RFC 7665, packet matching done by the proxies is not a trivial task. These elements need to match incoming and outgoing packets to re-insert the SFC encapsulation, or perform re-classification to add a new NSH header. However, the architecture does not impose any kind of restriction on the kind of changes on the packet allowed to service functions. These can go from header and payload modification to insertion and deletion of information and even creation of extra new packets. This generic nature turns packet matching into a challenging task, often requiring complex algorithms [Qazi et al., 2013b], that although functional are not fully precise. By moving these actions to the SF's execution unit's kernel, for example, they are executed on the same software domain, i.e., the same operating system.

Although beyond the scope of this work, in this setting packet matching could be facilitated by the use of structures offered by the OS itself. For example, if a zero-copy mode for packet handling is employed, matching single incoming and outgoing packets out become trivial, as matching packets would correspond to the same memory address. Other possibility would be to use packet metadata created by the kernel to identify packets and support correspondence between them.

Chaining-Box can be extended to support different types of functions by implementing function-specific Dec and Enc stages. Since they can be easily modified and reloaded to the kernel, one could apply different packet matching techniques based on prior knowledge on a specific function, i.e. about its behavior. In this case, different version of Dec and Enc stages would be loaded depending on what kind of function was being used. These changes would be limited to the service plane, not requiring changes to functions, keeping Chaining-Box's primary goal of being transparent.

Greater resilience to failure: as discussed in chapter 4, since Chaining-Box proposes a better distribution of SFC actions, there is a reduction on single points of failure that were concentrated on the Forwarders on the proposal of RFC 7665. That architecture relied on several interconnected star-like groups centered on Forwarders, with the consequence that a failure on one of the central points would affect all functions connected to it and the chains they belonged to. Chaining-Box allows the service plane to use more generic logical topologies, removing the centralized nodes. In this case, the failure of a single node disturbs only the chains that particular instance is part of.

Smaller forwarding tables: an extra consequence of each SF instance having its own dedicated Fwd stage is that the tables used by it for SFC forwarding will have less entries, as only the ones concerning the associated instance will be needed. This contrasts with the forwarding tables used by Forwarders in RFC 7665, which needed entries for several different instances attached to that particular element.

7.1.2 Benefits offered by BPF

Programmable stages: by nature BPF programs can be installed atomically by the kernel and have their functionality modified at any moment during run time. Its language is also generic enough to implement a wide range of programs. This makes the processing stages highly flexible and programmable. For example, a different packet matching technique can be implemented as a new pair of Dec and Enc stages and loaded instead of the standard programs, without knowledge from network and service function. The same could be applied to use new SFC or transport encapsulations or even change how the Fwd stage implements the forwarding functionality.

Offload to programmable devices: there are devices capable of executing BPF programs in hardware [Kicinski and Viljoen, 2016, Pacífico et al., 2018], and also native driver support offered by Linux to offload programs to them. This allows even better performance than what is obtained when running the programs inside the kernel.

Low resource usage: as shown previously, different from DPDK, which requires dedicated resources to execute at high speeds, BPF programs on XDP and TC allow similar performance at a considerable lesser memory and processor footprint.

7.2 Disadvantages

The remaining need for a classifier: even though the architecture is transparent to network and SFs, it still relies on a classifier to add the NSH header to each packet ingressing the SFC Domain, which will require some cooperation from the underlying network to either direct all packets to a few hosts running a software Classifier or implement such functionality on a border router, for example. Even with this remaining restriction, Chaining-Box represents a significant improvement towards a completely infrastructure-independent SFC architecture.

Dependence on encapsulation: the use of an extra encapsulation for SFC has a performance penalty caused by the constant encapsulation and decapsulation operations when using NSH-unaware functions. This may also cause fragmentation issues. However, this proposal aims for deployment on environments where an administrator has full power over the entire infrastructure, such as inside service provider datacenter networks. On such situations, the provider can configure the MTU of intermediate devices to a proper value considering the additional encapsulation, in order to avoid problems with packet fragmentation.

Implementation restricted to Linux: the proposed implementation based on BPF has the disadvantage of being limited to Linux environments, as the recent changes to BPF system are only available on that platform. The FreeBSD community has discussed about supporting the improvements offered by eBPF [1], but at the time of writing this has not been fully implemented yet. However, this is a minor restriction since most cloud and datacenter environments are Linux-based, thus able to support Chaining-Box if a more recent kernel version is used.

Lack of support for KB type functions: as discussed in §5.3 Chaining-Box currently does not support kernel-bypass functions, even if using AF_XDP sockets. An alternative approach would be to implement a BPF VM apart from the kernel, which could be used to create ingress and egress hooks for DPDK applications, for example, to which the processing stages could be loaded. Standalone implementations like this already exist, such as the BPF library on DPDK and the *ubpf* project [ubpf, 2019].

Packet matching: since the architecture is based on an external encapsulation, it requires matching packets, which is a difficult task. On the prototype implemented, this is done by performing an exact match on the packet's 5-tuple. This approach restricts the range of supported applications to only those that do not alter the packet's 5-tuple. However, as the stages are fully-programmable, other techniques can be used to implement Dec and Enc stages and change such behavior.

7.3 Other issues

The following items represent existing pain points caused by the lack of feature support by existing technologies, and are not directly caused by Chaining-Box's design choices.

No XDP on TX: the lack of an XDP hook on the egress direction of network interfaces hinders Chaining-Box in two ways. First, it forces Enc and Fwd stages be implemented on TC-egress. If not properly configured, other TC filters and actions can be executed after these stages, potentially affecting SFC functionality. By using an XDP hook, it would be more likely that these two last processing stages were some of the last pieces of code operating on the outgoing packet. The second consequence is that no BPF programs can be executed on egress packets from AF_XDP sockets, thus kernel-bypass functions are not supported by Chaining-Box.

No chained XDP execution: currently XDP programs cannot be executed in

[1]https://www.bsdcan.org/2018/schedule/track/Hacking/963.en.html

sequence, as is supported for TC programs using priorities and chains, for example. Thus each interface can only have a single XDP program attached to it. Chained execution can be emulated by the use of tail calls, but requires some changes to programs. Thus Chaining-Box currently cannot operate with service functions running on the XDP layer without changes to their source code.

Blackbox service functions: some existing service functions are implemented as black boxes, not allowing changes to its execution unit (be it VMs or containers). This makes it impossible to load the BPF processing stages, requiring the use a different technology for implementing the architecture.

Chapter 8

Conclusion

The main inquiry surrounding this work was *How to provide SFC in a fully transparent manner* while also meeting portability, reconfigurability and scalability requirements. We proposed Chaining-Box, an SFC architecture fulling these requirements while also providing extra benefits.

It is capable of operating in a fully transparent manner by placing all SFC functionality inside the Linux kernel in the form of BPF programs, not requiring any cooperation or knowledge from neither the service functions nor from the network infrastructure. The use of NSH encapsulation to create a service overlay further decouples it from the network, using the latter solely for packet forwarding based on well established protocols.

The use of BPF technology allows Chaining-Box to be readily deployable to any server with a recent Linux kernel, and leverages the kernel's support for runtime loading of BPF programs to be flexible to possible changes in chaining configuration and demand. All this with very low overhead, allowing Chaining-Box to be scalable to many CBox instances deployed on the same physical server.

The feasibility and different performance aspects of such approach have been evaluated, showing that although not yet on par with existing kernel-bypass or hardware-based alternatives, it is a promising solution providing a greater level of flexibility and independence from the network infrastructure.

8.1 Future Work

There are many aspects of the architecture still to be explored. For example, tests with actual service functions were out of the scope of this work, but are important demonstrations of how Chaining-Box can handle real applications and should be done in the future. Also different proxy strategies used by Dec and Enc stages could allow it to lift the restriction of supporting only functions that do not alter packet headers.

Another existing limitation is that Chaining-Box is currently only capable of handling linear chains. Support for branches and parallel execution with merging is also left as future work.

Furthermore, the overall performance could be improved by combining Enc and Fwd stages to avoid the extra table lookups and also offloading the stages to a programmable SmartNIC to be executed in hardware.

Lastly, new performance tests with less overhead inherent to the underlying setup would help understanding the architecture's own overheads, possibly helping optimizing it to better handle higher traffic loads.

8.2 Publications

During the development of this thesis, the author published a paper on the predecessor of Chaining-Box [Castanho et al., 2019], and was a key contributor to tutorials on how to use eBPF and XDP for fast packet processing, in order the share the learnings obtained from this work with the wider community [Vieira et al., 2019, Vieira et al., 2020]. He also helped developing a BPF loader and driver to interact with an FPGA-based serverless platform leveraging BPF [Pacífico et al., 2020].

Bibliography

[Abdelsalam et al., 2017] Abdelsalam, A., Clad, F., Filsfils, C., Salsano, S., Siracusano, G., and Veltri, L. (2017). Implementation of virtual network function chaining through segment routing in a linux-based NFV infrastructure. In *2017 IEEE Conference on Network Softwarization: Softwarization Sustaining a Hyper-Connected World: en Route to 5G, NetSoft 2017*, pages 1--5. IEEE.

[Ahmed et al., 2018] Ahmed, Z., Alizai, M. H., and Syed, A. A. (2018). Inkev: In-kernel distributed network virtualization for dcn. *SIGCOMM Comput. Commun. Rev.*, 46(3):4:1--4:6. ISSN 0146-4833.

[Beckett et al., 2018] Beckett, D., Joubert, J., and Horman, S. (2018). Host dataplane acceleration (hda).

[Bertin, 2017] Bertin, G. (2017). Xdp in practice: integrating xdp into our ddos mitigation pipeline. In *Netdev 2.1 - Technical Conference on Linux Networking*, pages 1--5.

[Burns et al., 2016] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *Queue*, 14(1):70--93.

[Castanho et al., 2018] Castanho, M. S., Dominicini, C. K., Villaça, R. S., Martinello, M., and Ribeiro, M. R. N. (2018). Phantomsfc: A fully virtualized and agnostic service function chaining architecture. In *Computers and Communications (ISCC), 2018 IEEE Symposium on*. IEEE.

[Castanho et al., 2019] Castanho, M. S., Vieira, M. A. M., and Dominicini, C. K. (2019). Cadeia-aberta: Arquitetura para sfc em kernel usando ebpf. In *Anais do XXXVII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*, pages 763--776. SBC.

[Cilium, 2019] Cilium (2019). Cilium: Api-aware networking and security. `https://cilium.io/`. Accessed on 09/09/2019.

[DPDK, 2020] DPDK (2020). Data Plane Development Kit.

[Duchene et al., 2018] Duchene, F., Jadin, M., and Bonaventure, O. (2018). Exploring various use cases for ipv6 segment routing. In *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, SIGCOMM '18, pages 129--131, New York, NY, USA. ACM.

[Dumazet, 2011] Dumazet, E. (2011). A jit for packet filters. https://lwn.net/Articles/437981/.

[Facebook, 2018] Facebook (2018). Katran source code repository. https://github.com/facebookincubator/katran.

[Fayazbakhsh et al., 2014] Fayazbakhsh, S. K., Chiang, L., Sekar, V., Yu, M., and Mogul, J. C. (2014). Enforcing Network-Wide Policies in the Presence of Dynamic Middlebox Actions using FlowTags. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 2014)*, pages 543--546, New York, New York, USA. ACM Press.

[FDio, 2020] FDio (2020). FD.io - The Universal Dataplane. https://fd.io/.

[Filsfils et al., 2019] Filsfils, C., Dukes, D., Previdi, S., Leddy, J., Matsushima, S., and Voyer, D. (2019). Ipv6 segment routing header (srh). Internet-Draft draft-ietf-6man-segment-routing-header-26, IETF Secretariat. http://www.ietf.org/internet-drafts/draft-ietf-6man-segment-routing-header-26.txt.

[Garay et al., 2016] Garay, J., Matias, J., Unzilla, J., and Jacob, E. (2016). Service description in the NFV revolution: Trends, challenges and a way forward. *IEEE Communications Magazine*. ISSN 01636804.

[Halpern and Pignataro, 2015] Halpern, J. and Pignataro, C. (2015). Service function chaining (SFC) architecture. RFC 7665, IETF.

[Høiland-Jørgensen et al., 2018] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. (2018). 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*, CoNEXT '18, pages 54--66, New York, NY, USA. ACM.

[John et al., 2013] John, W. et al. (2013). Research directions in Network Service Chaining. In *2013 IEEE SDN for Future Networks and Services (SDN4FNS)*. ISSN 1063-6692.

[Jouet et al., 2015] Jouet, S., Cziva, R., and Pezaros, D. P. (2015). Arbitrary packet matching in openflow. In *2015 IEEE 16th International Conference on High Performance Switching and Routing (HPSR)*, pages 1–6. ISSN .

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

[Kitada et al., 2014] Kitada, H. et al. (2014). Service function chaining technology for future networks. *NTT Technical Review*, 12(8). ISSN 13483447.

[Koch et al., 2019] Koch, J., Spier, M., Gregg, B., and Hunter, E. (2019). Extending vector with ebpf to inspect host and container performance.

[Kohler et al., 2000] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. (2000). The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263--297.

[Linux, 2017] Linux (2017). net: xdp: support xdp generic on virtual devices. `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=d445516966dcb2924741b13b27738b54df2af01a`.

[Luigi, 2020] Luigi, R. (2020). The Netmap Project. http://info.iet.unipi.it/ luigi/netmap/.

[Marek Majkowski, 2019] Marek Majkowski (2019). Cloudflare architecture and how bpf eats the world.

[Marmol et al., 2015] Marmol, V., Jnagal, R., and Hockin, T. (2015). Networking in containers and container clusters. *Proceedings of netdev 0.1.*

[Martins et al., 2014] Martins, J., Ahmed, M., Raiciu, C., Olteanu, V., Honda, M., Bifulco, R., and Huici, F. (2014). ClickOS and the Art of Network Function Virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 459--473, Berkeley. USENIX Association.

[McCanne and Jacobson, 1993] McCanne, S. and Jacobson, V. (1993). The bsd packet filter: A new architecture for user-level packet capture. In *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*, USENIX'93, pages 2--2, Berkeley, CA, USA. USENIX Association.

[Medhat et al., 2017] Medhat, A. M., Taleb, T., Elmangoush, A., Carella, G. A., Covaci, S., and Magedanz, T. (2017). Service Function Chaining in Next Generation Networks: State of the Art and Research Challenges. *IEEE Communications Magazine*, 55(2):216--223. ISSN 01636804.

[Merkel, 2014] Merkel, D. (2014). Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2.

[Miano et al., 2019] Miano, S., Bertrone, M., Risso, F., Bernal, M. V., Lu, Y., Pi, J., and Shaikh, A. (2019). 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. ISSN null.

[Miano et al., 2018] Miano, S., Bertrone, M., Risso, F., Tumolo, M., Bernal, M. V., and Tumolo, M. (2018). Creating Complex Network Services with eBPF: Experience and Lessons Learned. *High Performance Switching and Routing (HPSR). IEEE*, pages 1--8.

[Mijumbi et al., 2016] Mijumbi, R., Serrat, J., Gorricho, J. L., Bouten, N., De Turck, F., and Boutaba, R. (2016). Network function virtualization: State-of-the-art and research challenges. *IEEE Communications Surveys and Tutorials*, 18(1):236--262. ISSN 1553877X.

[Miller, 2017] Miller, D. (2017). Bpf verifier overview. `https://lwn.net/Articles/794934/`. Accessed on 09/04/2019.

[OpenDaylight, 2020] OpenDaylight (2020). OpenDaylight. https://www.opendaylight.org/.

[Openstack, 2020] Openstack (2020). What is OpenStack? https://www.openstack.org/software/.

[OVS, 2020] OVS (2020). Open vSwitch. https://www.openvswitch.org/.

[Pacífico et al., 2018] Pacífico, R. D., Coelho, G. R., Vieira, M. A. M., and Nacif, J. A. (2018). Roteador sdn em hardware independente de protocolo com análise, casamento e ações dinâmicas. In *Simpósio Brasileiro de Redes de Computadores (SBRC)*, volume 36.

[Pacífico et al., 2020] Pacífico, R. D., Duarte, L. F. S., Castanho, M. S., Nacif, J. A. M., and Vieira, M. A. M. (2020). Sistema de processamento de pacotes serverless. In *Anais do XXXVIII Simpósio Brasileiro de Redes de Computadores e Sistemas Distribuídos*. SBC.

[Panda et al., 2016] Panda, A., Han, S., Jang, K., Walls, M., Ratnasamy, S., and Shenker, S. (2016). Netbricks: Taking the v out of {NFV}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 203--216.

[Pfaff et al., 2015] Pfaff, B., Pettit, J., Koponen, T., Jackson, E., Zhou, A., Rajahalme, J., Gross, J., Wang, A., Stringer, J., Shelar, P., et al. (2015). The design and implementation of open vswitch. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 117--130.

[Pollitt, 2019] Pollitt, A. (2019). Tigera adds eBPF support to Calico . `https://www.projectcalico.org/tigera-adds-ebpf-support-to-calico/`. Accessed on 09/09/2019.

[Qazi et al., 2013a] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. (2013a). SIMPLE-fying middlebox policy enforcement using SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):27--38. ISSN 01464833.

[Qazi et al., 2013b] Qazi, Z. A., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., and Yu, M. (2013b). SIMPLE-fying middlebox policy enforcement using SDN. *ACM SIGCOMM Computer Communication Review*, 43(4):27--38. ISSN 01464833.

[Quinn et al., 2018] Quinn, P., Elzur, U., and Pignataro, C. (2018). Network service header (NSH). RFC 8300, IETF.

[Quinn and Guichard, 2014] Quinn, P. and Guichard, J. (2014). Service function chaining: Creating a service plane via network service headers. *Computer*, 47(11):38--44. ISSN 00189162.

[Quinn and Nadeau, 2015] Quinn, P. and Nadeau, T. (2015). Problem Statement for Service Function Chaining. RFC 7498, IETF.

[Sahhaf et al., 2015] Sahhaf, S. et al. (2015). Network service chaining with optimized network function embedding supporting service decompositions. *Computer Networks*, 93:492--505. ISSN 13891286.

[Tu et al., 2017] Tu, C., Stringer, J., and Pettit, J. (2017). Building an extensible open vswitch datapath. *SIGOPS Oper. Syst. Rev.*, 51(1):72--77. ISSN 0163-5980.

[ubpf, 2019] ubpf (2019). Userspace eBPF VM. `https://github.com/iovisor/ubpf`.

[Vieira et al., 2020] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D., Santos, E. R., Júnior, E. P. C., and Vieira, L. F. (2020). Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)*, 53(1):1--36.

[Vieira et al., 2019] Vieira, M. A. M., Castanho, M. S., Pacífico, R. D. G., Santos, E. R. S., Câmara Júnior, E. P. M., and Vieira, L. F. M. (2019). Processamento Rápido de Pacotes com eBPF e XDP.

[VPP, 2020] VPP (2020). Vector Packet Processor. https://fd.io/docs/vpp/master/.

[WeaveWorks, 2017] WeaveWorks (2017). Improving performance and reliability in Weave Scope with eBPF . `https://www.weave.works/blog/improving-performance-reliability-weave-scope-ebpf/`.

[Wiles, 2016] Wiles, K. (2016). pktgen DPDK.

[Xhonneux et al., 2018] Xhonneux, M., Duchene, F., and Bonaventure, O. (2018). Leveraging ebpf for programmable network functions with ipv6 segment routing. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 67--72, New York, NY, USA. ACM.

[Xia et al., 2015] Xia, M. et al. (2015). Optical service chaining for network function virtualization. *IEEE Communications Magazine*, 53(4):152--158. ISSN 01636804.

[Zhang et al., 2016] Zhang, W., Liu, G., Zhang, W., Shah, N., Lopreiato, P., Todeschi, G., Ramakrishnan, K. K., and Wood, T. (2016). OpenNetVM: A platform for high performance network service chains. In *HotMiddlebox 2016 - Proceedings of the 2016 ACM SIGCOMM Workshop on Hot Topics in Middleboxes and Network Function Virtualization, Part of SIGCOMM 2016*.

[Zhang et al., 2013] Zhang, Y., Beheshti, N., Beliveau, L., Lefebvre, G., Manghirmalani, R., Mishra, R., Patneyt, R., Shirazipour, M., Subrahmaniam, R., Truchan, C., and Tatipamula, M. (2013). StEERING: A software-defined networking for inline service chaining. In *Proceedings - International Conference on Network Protocols, ICNP*. ISSN 10921648.

Appendix A

Configuration file structure

As briefly discussed in Section 5.6, the configuration file given as input to the controller is expressed in JSON format. This file must contain two mandatory fields, as discussed in the following sections.

A.1 chains

This field is an array of chain objects representing all the chains that should be configured. Each element in the array is an object with two mandatory fields: `id` and `nodes`. The first is a simple integer indicating the SPI for that specific chain, while the second comprises of an array of strings with the names (`tag`s) of the functions that compose the chain.

Note that the names listed on the `nodes` field of each chain object must have a corresponding object with that tag in the `functions` array field discussed in the next section. Otherwise, the controller rejects the configuration file as it cannot properly configure the given list of chains.

A.2 functions

This field is an array of function objects used to tell the controller of all the SFs that will be part of the chains listed on the `chains` field. Each function object is composed of three mandatory fields:

- `tag`: this is the name by which this SF should be referred to by the controller. It is used to uniquely identify this instance among all others.

- `type`: this is an implementation-specific detail of the current prototype. It is used to indicate which of a set of known service functions should run on that specific node. On a more generic implementation, this should point to some way to deploy that specific function on the corresponding host.

- `host`: this field indicates the name of host on which this SF is hosted. It is mainly used by the controller to determine if direct links can be used. When two consecutive nodes in a chain are deployed to the same host, they are automatically connected by direct links.

A.3 Example

Listing A.1 gives yet another example of a valid Chaining-Box configuration file:

Listing A.1: Example of JSON configuration file

```json
{
    "chains": [
        {
            "id": 7892,
            "nodes": ["sf1"]
        },
        {
            "id": 1234,
            "nodes": ["sf1","sf2"]
        }
    ],
    "functions": [
        {
            "tag": "sf1",
            "type": "tc-redirect",
            "host": "server1"
        },
        {
            "tag": "sf2",
            "type": "tc-redirect-delay",
            "host": "server2"
        }
    ]
}
```

A.4 JSON Schema

The JSON Schema shown in Listing A.2 formalizes the structure of the file.

Listing A.2: JSON Schema of chaining configuration file

```
{
    "definitions": {
        "Configfile": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
            "chains": {
                "type": "array",
                "items": {
                    "$ref": "#/definitions/Chain"
                }
            },
            "functions": {
                "type": "array",
                "items": {
                    "$ref": "#/definitions/Function"
                }
            }
        },
        "required": [
            "chains",
            "functions"
        ],
        "title": "Configfile"
    },
    "Chain": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
            "id": {
                "type": "integer"
            },
            "nodes": {
                "type": "array",
                "items": {
                    "type": "string"
                }
            }
        },
        "required": [
            "id",
            "nodes"
        ],
        "title": "Chain"
    },
    "Function": {
        "type": "object",
        "additionalProperties": false,
        "properties": {
            "tag": {
                "type": "string"
            },
            "type": {
                "type": "string"
            },
            "host": {
                "type": "string"
            }
        },
        "required": [
            "host",
            "tag",
            "type"
        ],
        "title": "Function"
    }
}
```