卒業論文 2023 年度（令和 5 年度）

Linux Netfilter の SRv6 への統合

慶應義塾大学 環境情報学部
澤田 開杜

卒業論文要旨 - 2023 年度（令和 5 年度）

Linux Netfilter の SRv6 への統合

本論文では，Linux の持つパケット操作機能である netfilter を，トラフィック制御技術の 1 つである SRv6 に統合する手法を提案する．昨今のデータセンタネットワークでは，汎用的なサーバや仮想マシン，コンテナ技術を使ってネットワークの機能 (NF) を仮想化する技術が一般化してきている．一連のルールに沿って NF を適用することを サービスファンクションチェイニング (SFC) という．SFC にデプロイされる NF はサービスファンクション (SF) と呼ばれ，SFC ではあるパケットを任意の順番で SF へ通過させる必要がある．従来のパケットルーティングでは，あるパケットを任意の順番で指定したノードを通過させる，ということはでききない．よって，SFC の実現のためには従来のパケットルーティングとは別の経路制御機構が必要である．SFC を実現できる経路制御技術の 1 つに Segment Routing over IPv6 (SRv6) という技術がある．SRv6 では，SRv6 header と呼ばれるヘッダで IP パケットをカプセル化する．また，SRv6 header にはパケットが通過するノードが順番に含まれる．これによって，IP 的なベストパスに関係なくパケットが通過するノードを指定可能であるから，任意のルールに従って SF を通る順番を指定できる．また，SRv6 はトラフィック制御だけでなく，トランジットするパケットに対して特定の操作を適用でき，この特定の操作の種類のことを**ビヘイビア**という．

Linux カーネルには netfilter というパケット処理フレームワークが実装されている．netfilter を使うことで，パケットのフィルタリングや NAT，NAPT，その他のパケットマングリング操作を適用できる．しかし，SRv6 は SRv6 header でカプセル化されているため，カプセル化されている内部のパケットに対して netfilter を適用できない．そこで，本論文では End.AN.NF という新しい SRv6 ビヘイビア を提案する．End.AN.NF は Linux netfilter を Linux の SRv6 ルーティングインフラストラクチャへ統合する事ができる．End.AN.NF は，SRv6 を利用した SFC 環境において，Linux netfilter を SRv6 に対応した SF として扱えるようにする．End.AN.NF を利用する際，netfilter を利用して作成されたアプリケーションの実装を変える必要はなく，End.AN.NF は SRv6 の基本処理である End ビヘイビアを実行しながら，SRv6 でカプセル化された内部のパケットへ netfilter を適用できる．さらに，End.AN.NF は，パケットバッファにマークを付けることができる．したがって，netfilter を利用して作成されたアプリケーションは End.AN.NF がパケットバッファに付与したマークを照合することで，適用するルールを変更できる．我々は End.AN.NF を Linux カーネルに実装し，その性能評価を行った．計測の結果，End.AN.NF は End.DT4 と H.Encaps を使って SRv6 でカプセル化された内部パケットに netfilter を適用する方法に比べ，27% 高いスループット，及び 3.0 マイクロ秒低いレイテンシを実現した．

キーワード:
1. Service Function Chaining 2. Segment Routing 3. SRv6

慶應義塾大学 環境情報学部
澤田 開杜

Abstract of Bachelor's Thesis - Academic Year 2023

Integrating Netfilter into SRv6 Routing Infrastructure of Linux as an SR-Aware Network Function

This paper proposes a new SRv6 End behavior, called End.AN.NF, integrating Linux netfilter as a network function for service function chaining by Segment Routing (SR). End.AN.NF allows netfilter-based applications to be executed as SR-Aware applications without modification, as it applies netfilter to inner packets encapsulated in SRv6 while performing the basic SRv6 End behavior. Furthermore, End.AN.NF utilizes the argument of the segment identifiers to mark packets. Consequently, this enables netfilter-based applications to match the marks on packet buffers and change rules to be applied. We implemented End.AN.NF on the Linux kernel and evaluated its performance. The evaluation shows that End.AN.NF achieves 27% higher throughput and 3.0 microseconds lower latency than applying netfilter to SRv6-encapsulated inner packets by End.DT4 and H.Encaps.

Keywords :
1. Service Function Chaining 2. Segment Routing 3. SRv6

Keio University Bachelor of Arts in Environment and Information Studies
Kaito Sawada

目 次

第 1 章　序論 **1**
1.1 Service Function Chaining 1
1.2 従来のパケットルーティングとトラフィックエンジニアリング 2
1.3 SRv6 6
1.4 導入 10
1.5 本論文の目的と構成 12

第 2 章　背景と問題提起 **13**
2.1 Linux と netfilter 13
2.2 SRv6 と SF としての netfilter 統合手法 14
2.3 問題提起 17

第 3 章　提案手法の設計と実装 **18**
3.1 提案手法 18
3.2 設計 18
3.3 実装 21
3.3.1 Linux における SRv6 ビヘイビアの実装 21

第 4 章　評価 **26**
4.1 計測の概要と予想 26
4.2 TRex 28
4.3 レシーブサイドスケーリング (RSS) 28
4.4 パケットサイズ毎のスループット性能 30
4.4.1 計測内容 30
4.4.2 評価 31
4.5 netfilter にインストールされたルール毎のスループット 31
4.5.1 nftables 32
4.5.2 計測内容 35
4.5.3 評価 35
4.6 レイテンシ 36
4.6.1 計測内容 37
4.6.2 評価 38

第 5 章　結論 **39**

謝辞 **40**

図 目 次

1.1 SFC Architecture 2
1.2 Explain Source Routing 3
1.3 Architecture of OpenFlow 5
1.4 Structure of NSH 6
1.5 Architecture of SRv6 7
1.6 layer-3 VPN with SRv6 8
1.7 SRv6 Packet Structure 10

2.1 netfilter hook points (wiki.nftables.org より引用 [1]) 14
2.2 SR-proxy Architecture 16

3.1 `End.AN.NF` applies three netfilter hook points, prerouting, forward, and postrouting, to inner packets encapsulated in SRv6. 20
3.2 The modified Linux kernel treats an `End.AN.NF` SID as an IPv6 routing table entry. We can manage the `End.AN.NF` routes with the existing tools such as iproute2. 20

4.1 The difference between "End.DT4 and H.Encaps" and End.AN.NF 27
4.2 Abstract of DPDK 29
4.3 Abstract of RSS 30
4.4 Throughput per SRv6 End behaviors and IPv4 32
4.5 Throughput per number of rules of base chains 36
4.6 Throughput per number of rules of regular chains 37
4.7 Latency per SRv6 End behaviors and IPv4 38

表 目 次

第1章　序論

1.1　Service Function Chaining

SFC とは，エンドツーエンド通信を提供するために必要なさまざまなサービスファンクション (SF) を決定及び順序付けし，それらを介するようにトラフィックを操作することを指す．図 1.1 に，SFC アーキテクチャの概略図を示す．SF には，ファイアウォールや IP ネットワークアドレストランスレータ (NAT) などのネットワークサービスファンクションや，アプリケーション固有の機能が含まれる．SFC アーキテクチャは，基礎となるネットワークトポロジから独立したトポロジを前提としている．この基礎となるネットワークトポロジをアンダーレイネットワークといい，独立した SFC のためのネットワークトポロジをオーバレイネットワークという．図 1.1 では灰点線のパスが物理的なパスであるアンダーレイネットワークを示し，その他の実線が SFC として選択可能なパスの例であるオーバレイネットワークを示している．SFC アーキテクチャでは，パケットは通信の入口となるノードで事前に定義されたポリシとパケット内の情報から分類され，SFC 対応ドメイン内で適用する SF のセットを決める．その後，任意の順番で各 SF でパケット処理が適用されるように転送される．例えば，Logging をした後 FW Service を適用，Filtering を適用すような場合は，図 1.1 における緑のパスを辿るようになる．

SFC アーキテクチャはネットワークの用途や運用計画などのコンテキストに依存しない汎用的な場面で利用可能な技術であり，SFC アーキテクチャは固定ネットワークやモバイルネットワーク，多くのデータセンターアプリケーションに適用できる．SFC の構築に関して，すべての SF が満たさなければならない標準の定義や特性は存在しない．各 SF は単に「パケットに対する特定の処理を適用できる要素」として扱われ，特定のネットワークで常に有効な SF を静的に列挙することはできない．なぜなら，適用する SF の集合はその瞬間に有効な SF であり，それらが有効かどうかはその時々のネットワーク環境によって異なる場合があるからである．SF のチェインとそれらを呼び出す基準は，SF 対応ドメインを運用する各ネットワーク毎に固有である．

SFC における SF は，受信したパケットの特定の処理を担当する機能である．SF はプロトコルスタックのさまざまなレイヤで動作し，論理的なコンポーネントとして仮想要素として実現されることもあれば，物理的な筐体としてネットワークの中に組み込まれることもある．近年では，SF が動作するマシンは物理的な筐体ではなく，汎用マシンにインストールされたハイパーバイザ上の VM の中で動作することも多い．また，コンテナ技術の台頭により，SF 自体をコンテナに閉じ込めてデプロイすることも一般的になっている．さらに，それらのコンテナを kubernetes [2] に代表されるコンテナオーケストレータによって管理する手法も提案されている [3] このように，仮想化されたネットワーク上の

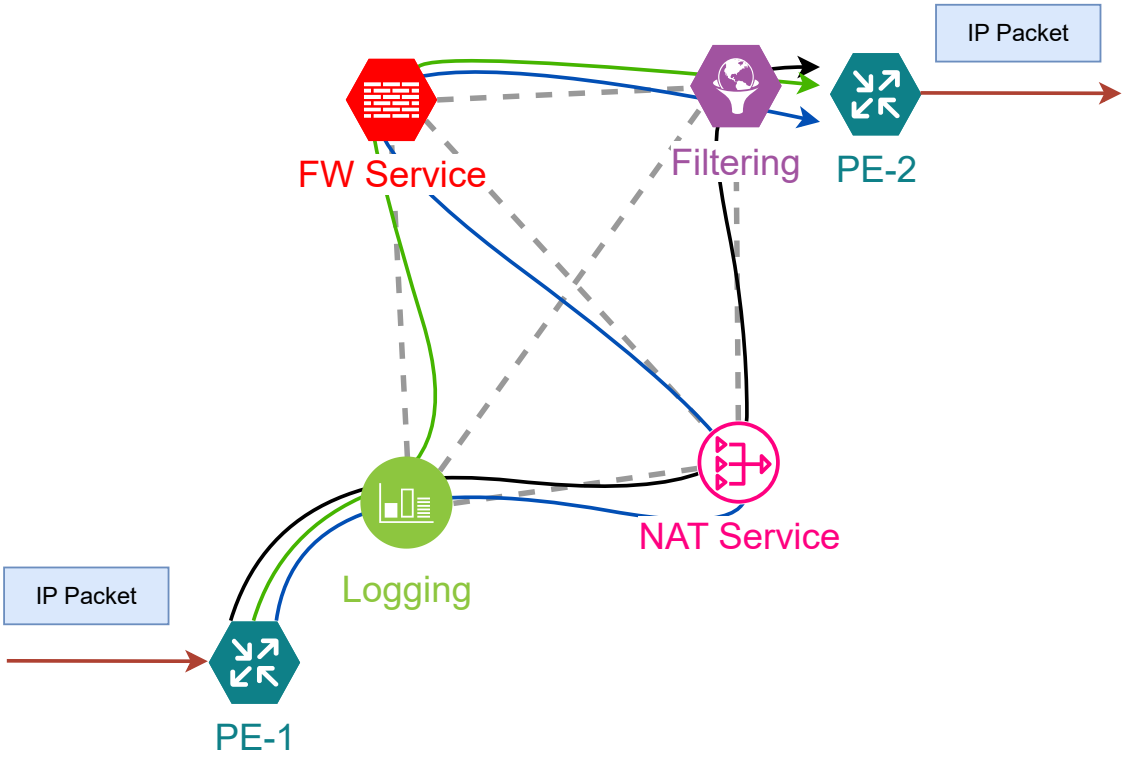

図 1.1: SFC Architecture

機能 (Network Function / NF) を NFV という．SFC は NFV の利用例として NFV のコンテキストでも研究されてきた技術である．

1.2 従来のパケットルーティングとトラフィックエンジニアリング

章 1.1 で述べた通り，SFC を実現するためには，エンドツーエンド通信を提供するために必要な複数の SF を決定，及び順序付けし，それらを介するようにトラフィックを操作する必要がある．しかし，従来のパケットルーティングでこのようなトラフィック操作を実現することは難しい．

ルータが IP パケットを転送するとき，ルータは自身の持つルーティングテーブルを参照する．このルーティングテーブルは通常，BGP [4] や OSPF [5]，IS-IS [6] などのルーティングプロトコルを通じて交換した経路情報から作成される．一般的に，多くのルーティングプロトコルでは「経由するノードの数を最小にする経路を最も良いものとする」という基本設計をもとに，オペレータが任意に決定したコスト情報などを含めて最も良い経路を計算する．このように決定された最も良い経路をベストパスといい，ルーティングテーブルには「ある宛先アドレスを持つパケットは次にどのノードに転送するのがベストパスなのか」が書かれている．ルータは，自身に接続されているノードやルーティングプロトコルを通じて受け取った経路情報が変更されたとき，その変更を近接ルータに通知

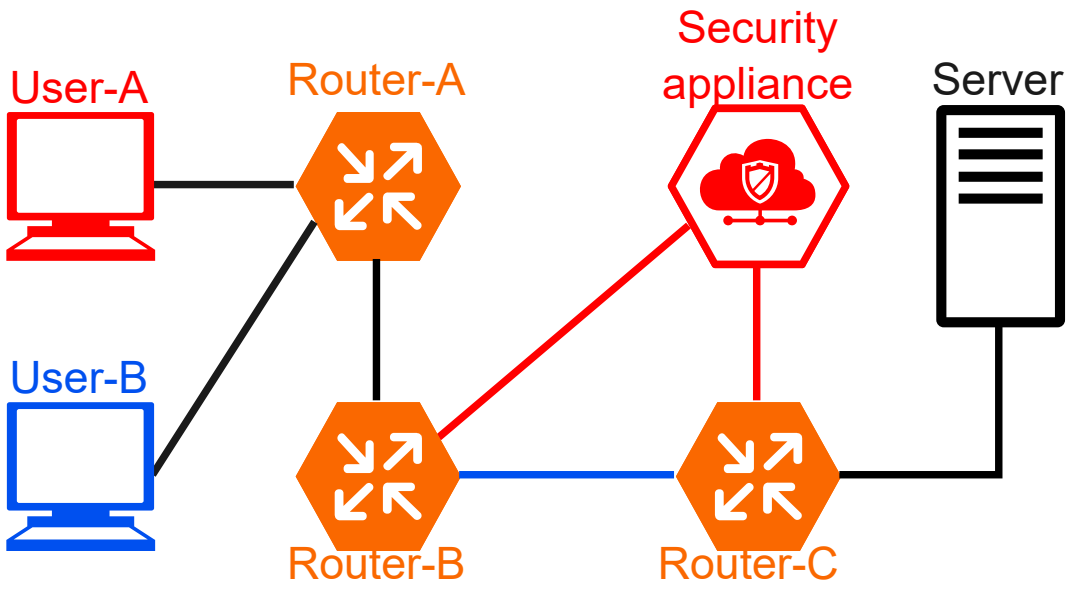

図 1.2: Explain Source Routing

し，自身のルーティングテーブルを更新する．ルーティングテーブルはオペレータが静的に構築することもできる．しかし，静的に経路を決定してしまうとノードの近接情報が変わるたびにオペレータ自身が設定し直す必要があり，これは手間がかかったりオペレーションミスを誘発したりする問題がある．そのため，静的な経路設定が利用される場面は限定的である．

図 1.2 に，あるネットワークのトポロジを示す．このネットワークにおいて，User-B の通信は青いパスを通るように，User-A から Server に向かう通信は Security appliance を経由させるような経路制御を行いたい．しかし，Router-B から Router-C までの経路は青いパスを通ると 1 hop だが，Security appliance を通る赤い経路は 2 hop である．つまり，Router-B から Server までの経路は青いパスの方が経由するノードの数が少ない．そのため，通常のルーティングプロトコルで経路を学習すると，Router-B のルーティングテーブルには Server に向かう経路として青いパスがベストパスとして採択される．ルーティングプロトコルの設定でコストを変更することで赤いパスをベストパスにすることは可能である．ベストパスを赤いパスに変更すると，User-A の通信は意図通り赤いパスを通るようになる．しかし，ベストパスを変更してしまうと User-B の通信についても赤いパスを通るようになり，これは意図した通信経路にならない．

ベストパスによらずに，また特定の通信毎に選択して経路を制御することを，トラフィックステアリング，トラフィックエンジニアリング (TE) という．TE が可能なパケット転送メカニズムとして，いくつかの候補が存在する．例えば OpenFlow [7]，Network Service Header (NSH) [8]，MPLS [9] などである．

OpenFlow

OpenFlow のによるネットワーク構成の概略図を 図 1.3 に示す．OpenFlow では，OpenFlow スイッチと呼ばれる OpenFlow に対応した専用のネットワーク機器をパケットを転送するノードとして使用し，OpenFlow コントローラと呼ばれる専用のマシンが OpenFlow スイッチの経路情報を集中管理する．OpenFlow のアーキテクチャは一般のパケットルーティングアーキテクチャとは大きく異なる．一般的なネットワークでは先に挙げた BGP や OSPF などのルーティングプロトコルを利用して経路情報を交換し，ルーティングテーブルを作成する．そして，ルータは自身が作成したルーティングテーブルに基づいてパケットを転送する．対照的に，OpenFlow では BGP や OSPF などのルーティングプロトコルを利用してルーティングテーブルを作成することはしない．経路情報は OpenFlow コントローラが集中管理し，OpenFlow コントローラは自身が決定した経路情報を実際にパケットを転送する OpenFlow スイッチへインストールする．また，OpenFlow ではルーティングテーブル自体も一般的なルーティングで作成されるものとは異なり，OpenFlow で利用されるルーティングテーブルに対応するテーブルのことをフローテーブルという．OpenFlow のフローテーブルには，宛先の IP アドレスだけではなく，送信元アドレスや通信を受信したポート，独自定義の専用パケットヘッダのフィールドなどもマッチングルールとして含まれる．これにより，IP 的なベストパスによらずに，かつ同じ宛先であっても別のパスを選択できる．

NSH

NSH は，SFC を実現する 1 つの手法として考えられたプロトコルで，NSH と呼ばれるヘッダでパケットをカプセル化する．OpenFlow が汎用的なパケット転送アーキテクチャとして考案されたのとは対象的に，NSH は SFC を前提として考案された．NSH で転送されるパケットの構造を図 1.4 として示す．NSH パケットは，大きく分けて 3 のパートに分けられる．Original Packet は，実際のエンドツーエンドでやり取りするパケットのことを指す．そのパケットを，NSH でカプセル化している．NSH の中にはいくつかのフィールドが存在し，その中には SFC の中でどのパスを通過するのか通過するのかや，ユーザ定義のメタデータ，オプショナルなフィールドなどが含まれる．NSH でカプセル化されたパケットを，更に Transport Encapsulation というヘッダがカプセル化している．この Transport Encapsulation は特定のフォーマットである必要はなく，GRE，VXLAN などの一般的なトンネリングプロトコルや通常のイーサフレームである．Transport Encapsulation の目的は，オーバーレイネットワークを通じて適切な SF ノードまでパケットを転送することである．NSH に対応したノードでは，SF が適用されたパケットに対して，そのパケットの NSH を参照し次の SF のノードを決定する．次の SF ノードが決まったらそのノードに届くよう，対応する Transport Encapsulation でカプセル化することで TE ができる．

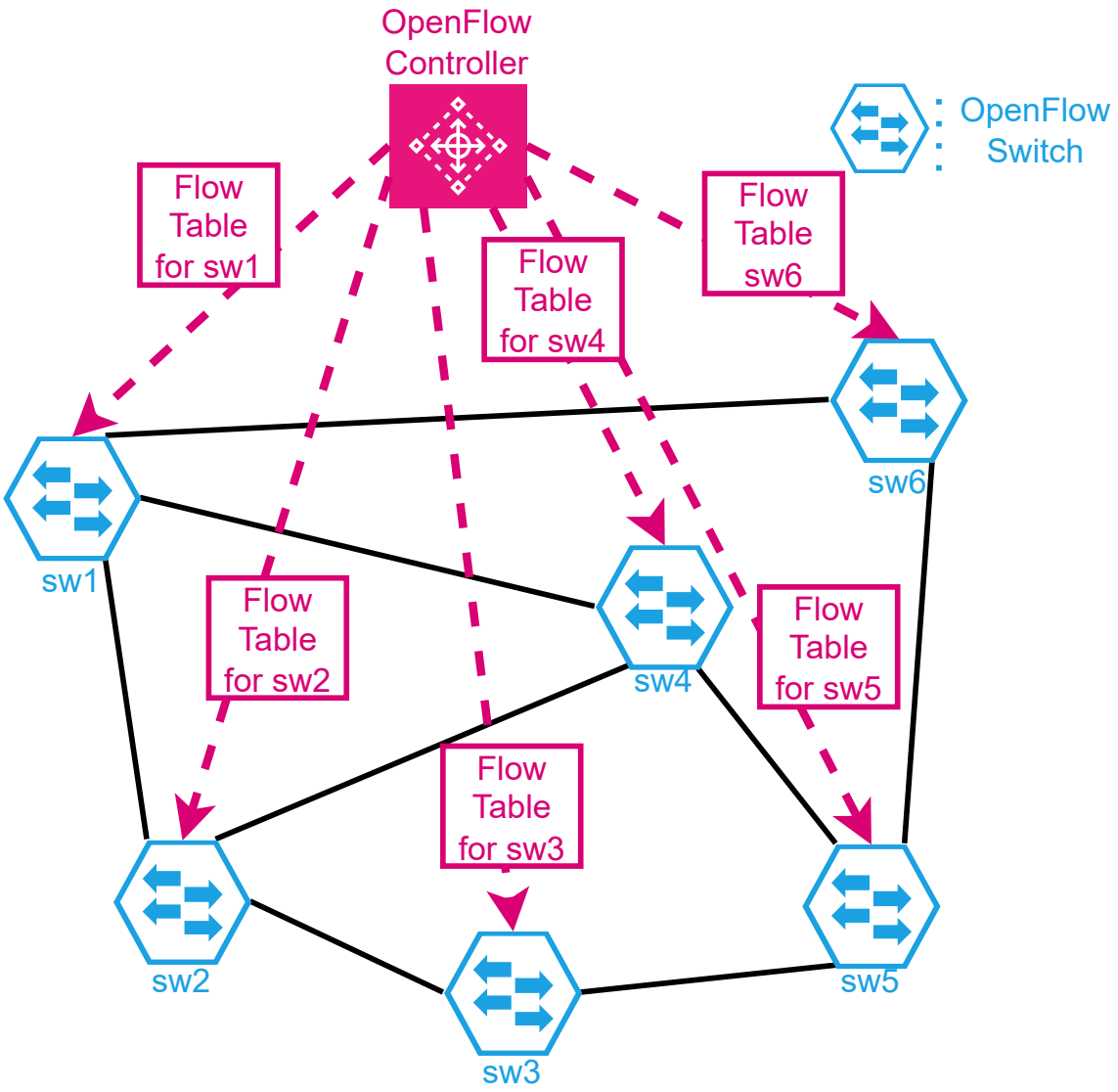

図 1.3: Architecture of OpenFlow

MPLS

MPLS とは Multiprotocol Label Switching の略称であり，MPLS ヘッダに含まれるラベル情報に基づいてパケットを転送するプロトコルである．MPLS ヘッダは Layer 2 ヘッダと Layer 3 ヘッダの間に挿入され，MPLS ヘッダの中にラベル情報を始めとするいくつかのフィールドが埋め込まれる．MPLS では，宛先の IP アドレスではなく，MPLS ヘッダに含まれるラベル情報を参照してパケットを転送する．つまり MPLS では，MPLS ヘッダ内に埋め込まれたタグ情報を使ってパケットを転送するため，IP 的なベストパスによらないルールでパケットを転送できる．また，MPLS は古くから VPN を構成するために用いられてきた一般的なプロトコルである．そのため，多くのネットワーク機器でサポートされている．OpenFlow は特別な機器やコントローラが必要であり，NSH も比較的新しいプロトコルで，かつ機能も SFC に特化しているため，NSH をサポートする機

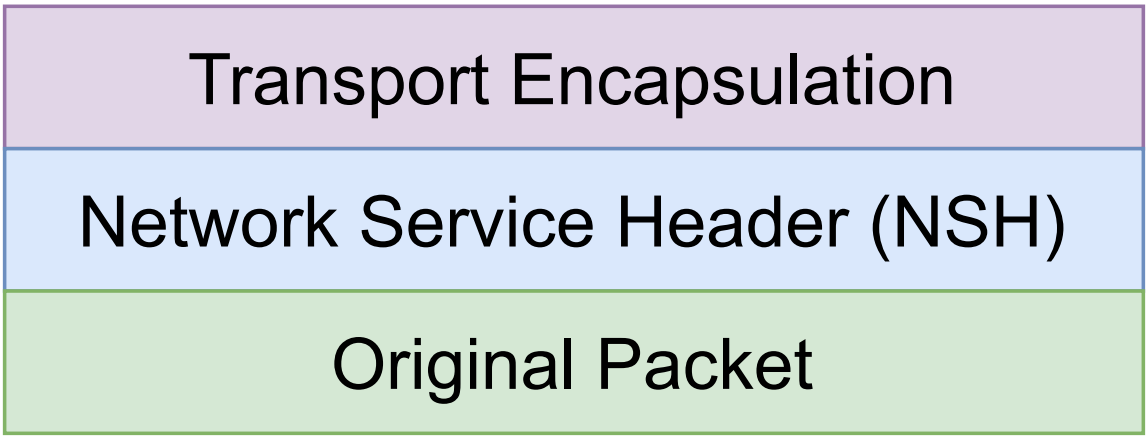

図 1.4: Structure of NSH

器は少ない．対象的に MPLS は既に多くの機器でサポートされているため，OpenFlow や NSH と比較して導入が容易であるという特徴を持つ．また，RFC8595 [10] のように，NSH と MPLS を組み合わせて SFC を実現する手法も提案されている．このアーキテクチャでは，NSH の Transport Layer として MPLS を利用している．

1.3 SRv6

先に挙げた技術だけではなく，Segment Routing over IPv6 (SRv6) も TE を適用できる技術の 1 つである．MPLS が独自のラベルを使って使ってパケットを転送するアーキテクチャであるのに対し，SRv6 では MPLS のラベルに対応する概念として IPv6 アドレスを利用する．SRv6 で利用される識別子はセグメント識別子 (SID) と呼ばれ，各 SID はネットワーク内の特定の場所で実行される特定の機能を表す．この SID は IPv6 と全く同じフォーマットをしている．SRv6 では，SRv6 ヘッダ (SRH) と呼ばれる IPv6 拡張ヘッダに SID の一連の集合からなるリストを埋め込むことで，ネットワークオペレータやアプリケーションはパケットが通過する中間地点を指定できる．

SRv6 ヘッダには通過するネットワーク上のノードの順番がリストとして埋め込まれ，ルータはそのリストに基づいてパケットを転送する．図 1.5 において，例えば SID リストの要素が **A，FW，C** である場合，パケットは緑色のパスを通るように転送される．また，SRv6 はあるパケットが経由するノードを指定できるだけでなく，パケットに対してパケットに対して特定の操作を適用できる．このようなパケット操作の種類のことを **SRv6 ビヘイビア** という．現在 RFC8986 [11] では 15 種類の End ビヘイビアが定義されている．

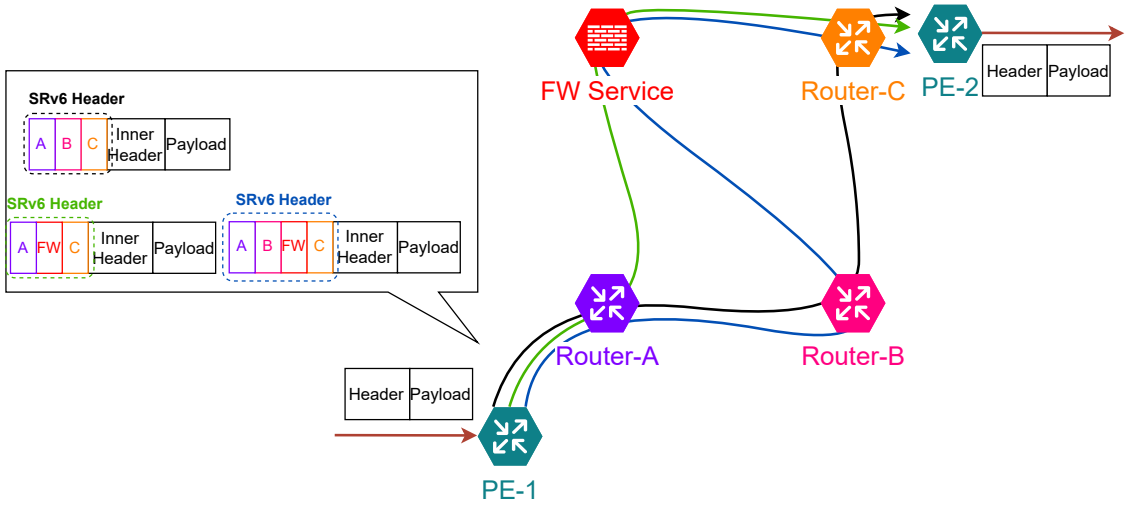

図 1.5: Architecture of SRv6

SRv6 を利用した layer-3 VPN の構築例と SRv6 によるパケット転送の具体的な動作

SRv6 ビヘイビアを組み合わせることで，layer-3 VPN を構成することもできる [12]. SRv6 を利用した layer-3 VPN の動作を図 1.6 に示す.

図 1.6 ① において，PE-1 は受信したパケットを SRH でカプセル化する．このように SRH でパケットをカプセル化する，という操作も SRv6 ではビヘイビアとして定義されており，この操作のことを H.Encaps という．ここでは，PE-1 は受信したパケットに対して **A，FW，C** を意味する SID リストを付加したものとする．このとき，SRv6 でカプセル化されたパケットの宛先アドレスは，内部パケットの宛先アドレスに関わらず，次に到達すべきノードを示す SID である FW Service になる.

PE-1 は H.Encaps で SRH を付加したパケットを Router-A へ送信する．このとき，パケットは宛先 IPv6 アドレスが Router-A である単なる IPv6 パケットとして扱われる．PE-1 は自身の持つルーティングテーブルを参照し，Router-A へのネクストホップを決定し，パケットを送出する．図 1.6 ② は，Router-A が受信したパケットに対して SID を 1 つ進めて FW Service にパケットを転送している様子を示している．リスト状になっている SID の中でどれが現在有効な SID であるかを指定するために，SRH にはセグメントレフト (segleft) と呼ばれるフィールドが定義されている．segleft は SID リストのインデックスであり，$(SID$ の合計$) - 1$ から始まり，0 で終わる.

Router-A は受信した SRv6 パケットの segleft を 1 つデクリメントし，FW Service の SID が次に有効な SID であることを示すようにする．また，Router-A はパケットの宛先アドレスを新しく有効になった FW Service の SID に書き換える．このように，segleft を 1 つ進め，宛先アドレスを新たに有効になった SID で書き換える動作のことを End ビヘイビアといい，これも SRv6 ビヘイビアの 1 つである．End ビヘイビアは SRv6 の中で最も基本的なビヘイビアである.

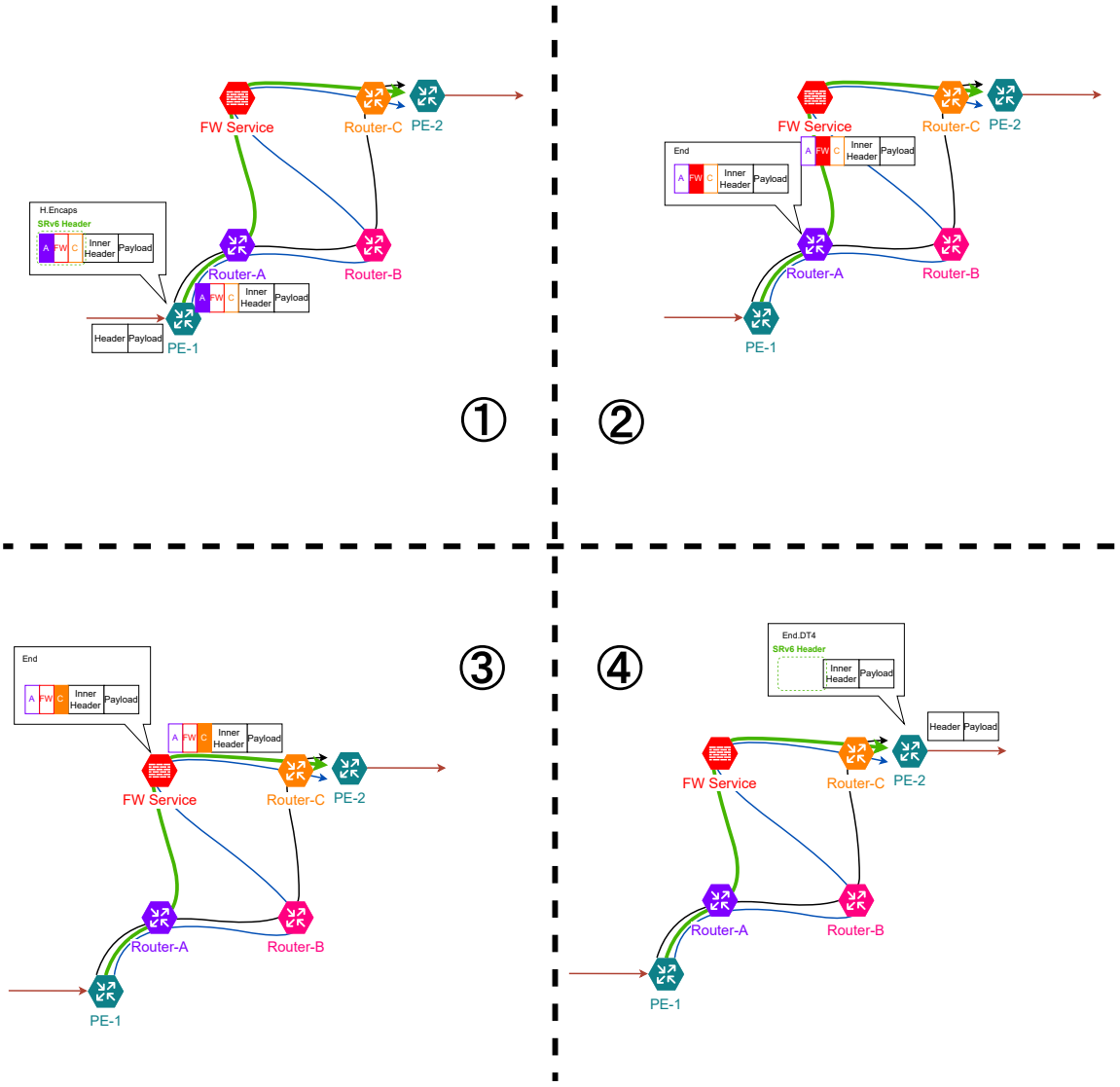

図 1.6: layer-3 VPN with SRv6

宛先アドレスを書き換えたあと，Router-A は自身のルーティングテーブルから新たな宛先アドレス (FW Service の SID) を検索し，ネクストホップへ転送する．図 1.6 ③ では，② と同様に FW Service が End ビヘイビアを実行して segleft デクリメントし，新たに有効になった SID に基づいて Router-C へパケットを転送している．図 1.6 ④ では，PE-2 が受信したパケットに対して End.DT4 というビヘイビアを実行している．このビヘイビアは，SRH を取り除き，特定の VRF を参照して SRv6 でカプセル化されていた内部パケットを転送する，という動作を実行する．End.DT4 により，パケットから SRH は取り除かれ，PE-1 でカプセル化される前のパケットを得ることができる．

SRv6 パケットと SID の構造

図 1.7 に，SRv6 パケットの詳細な構造を示す．図 1.6 では，簡単のためにパケット構造の図を簡略化して表現した．実際の SRv6 パケットは，一般的な IPv6 ヘッダの下に SRH が位置する．IPv6 ヘッダ内の Next Header (NH) には 43 が設定され，この 43 という数字は次に来るヘッダのタイプが IPv6 ルーティングヘッダであることを示す．

先に述べた通り，SID は SRv6 で利用される識別子で，SRH にはいくつかの SID がリスト状になって含まれている．現在どの SID が有効であるかは，segleft の値によって決まる．図 1.7 の例では，segleft は 1 である．よって，Segment List のインデックスが 1 である，最初から 2 番目の SID が有効化されている．宛先アドレスには現在有効な SID の値が設定されるため，このパケットの宛先アドレスは Segment List の 2 番目の SID となる．

図 1.7 で示されているように，SID は `LOC:FUNCT:ARG` という 3 つのパートに分かれている．また，SID は IPv6 アドレスであるため，それら 3 つパートの長さの合計は 128bit である．`LOC` はロケータを表す．ロケータとは，ある SID に対応するノードの場所を表す．`FUNCT` は SID に関連付けられた SRv6 ビヘイビアの識別子であり，`ARG` はビヘイビアの動作に必要な追加情報をエンコードする領域である．例えば，End.DT4 では `ARG` フィールドにはルックアップするべき VRF テーブルを識別するための情報がエンコードされる．

SRv6 の SID は IPv6 アドレスであるため，既存のルーティングプロトコルを用いて SID を経路情報として既存のルーティングプロトコルを利用して広告できる．SRv6 ビヘイビアは，SID の `FUNCT` パートで表現される．ただし，「特定の SRv6 ビヘイビアならば `FUNCT` パートはこの値である」というような一般的な定義は存在しない．例えば，IS-IS の TLV の Type フィールドの値は IANA によって策定されいる [13]．それに対して，SRv6 ではビヘイビアに対応する値は標準化されず，オペレータが自由に決めることができる．図 1.6 ④ では，Router-C は受信したパケットに対して End.DT4 を実行している．Router-C は `LOC:FUNCT:ARG` のフォーマットに従って [**Router-C のロケータを示すブロック**]:[**Router-C 上で End.DT4 として定義されたブロック**]:[**利用可能な VRF table 番号**] を自由に決定し，それを経路情報として周りのノードへ広告する．

SRv6 ビヘイビアが実装されていないネットワーク機器が SRv6 でカプセル化されたパケットを受信した際，その機器は SRv6 パケットを IPv6 パケットとして転送できる．SRH の前には一般的な IPv6 ヘッダが挿入されているため，そのパケットは IPv6 パケットとして転送可能であるからである．つまり，SRv6 ビヘイビアを実行せず，単にパケットを現在有効な SID へ転送するだけであれば，IPv6 パケットフォワーディングが実装されていれば適切にパケットを転送できる．

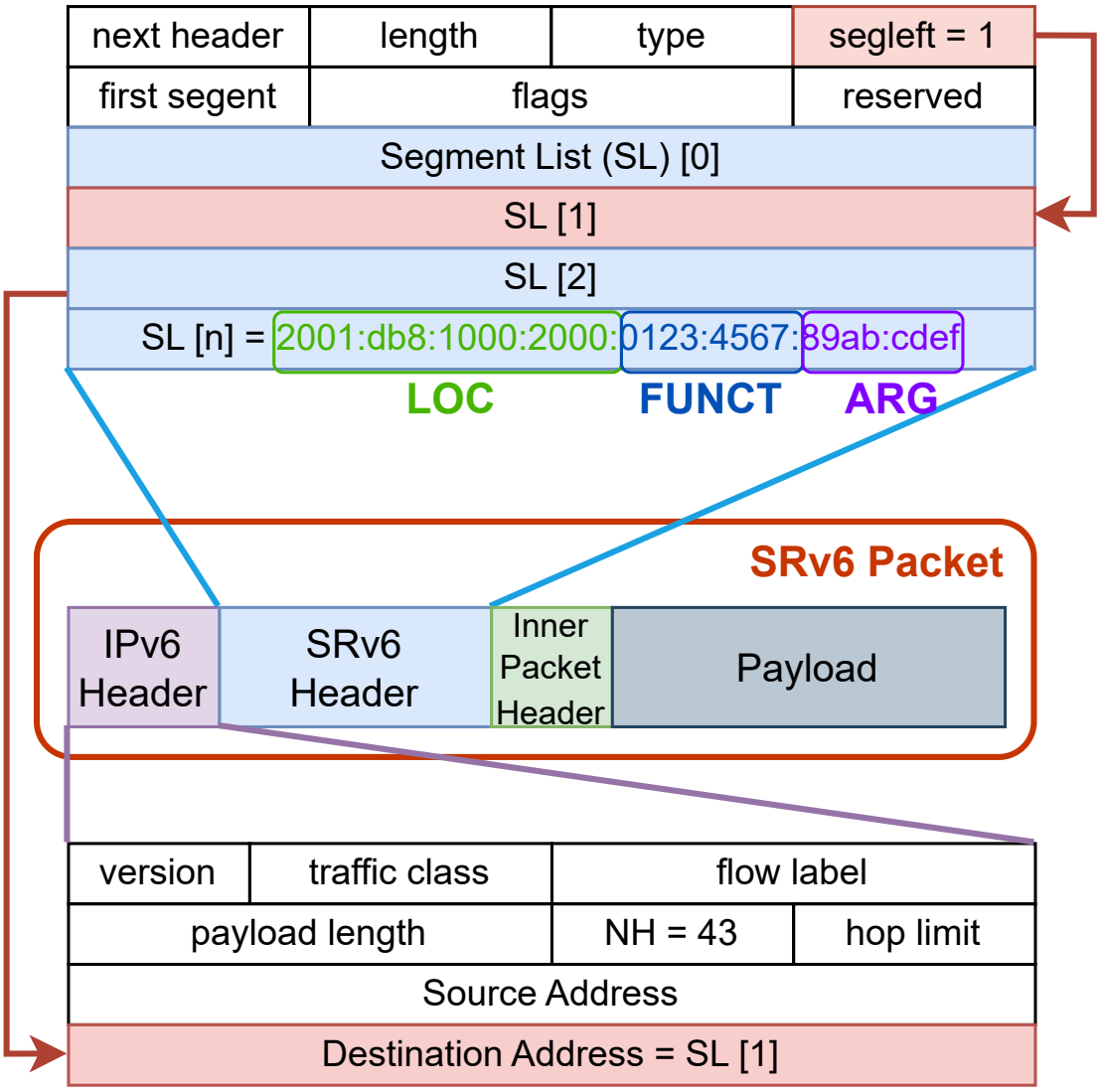

図 1.7: SRv6 Packet Structure

1.4 導入

Service Function Chaining (SFC) は，Software Defined Network (SDN) 及び Network Function Virtualization (NFV) の文脈で研究されているトピックである [14, 15, 16, 17]. SFC では，サービスファンクション (SF) を通過する順序や SF のタイプに関する情報を事前に定義し，それらのルールをネットワーク機器に配布する必要がある．SFC ネットワークを構築するネットワーク機器は，事前に決定されたルールに従って受信したパケットを SF に導く．パケットを NF へ導くためのルールは，SDN コントローラやルーティングプロトコルによってネットワーク機器に配布される．ネットワーク機器は IP ルーティング上の最短経路に関係なく，配布された SFC ルールに従ってパケットを転送する次のホップを選択する必要がある．また，パケットのヘッダにこれらのルールに合致させるための特別な情報を埋め込む手法が取られることもある．SFC は，クラウドサービスプロ

バイダ (CSP)，アプリケーションサービスプロバイダ (ASP) 及びインターネットサービスプロバイダ (ISP) にとって，現在の静的な環境に代わる柔軟かつ経済的な選択肢を提供する [18].

SFC を実現可能な技術には，いくつかの候補が存在する．例えば OpenFlow [7]，Network Service Header (NSH) [8]，MPLS [10] などである．これらの技術はどれも，最短経路に関係なく，ルールに基づいて受信したパケットを意図した SF に導く，という要件を満たすことができる．OpenFlow では，経路情報を管理する中央のコントローラが，実際にパケットを転送する OpenFlow スイッチに対して明示的にパケット転送ルールを設定する．OpenFlow スイッチは，コントローラによって適切に管理されたルールに従い，パケットを意図した SF に転送する．OpenFlow のもつこのアーキテクチャは，従来のルーティングプロトコルに基づかない柔軟な経路制御を可能にする．NSH は Service Path Identifier (SPI) と Service Index (SI) によって SF を識別する．NSH ノードは，パケットに付与された NSH 内の SPI，SI に基づいてパケットを転送する．NSH は，サービスプレーンと呼ばれる専用のオーバーレイネットワークを作成し，そのオーバレイネットワーク内でサービスを転送する．このオーバレイネットワークを構築する，というアーキテクチャにより，NSH では基礎となるネットワークトポロジを変更することなくサービス転送を可能にする．一方，MPLS では，直接 NSH を使用する代わりに，MPLS ラベルスタックを利用する．このラベルスタックには，パケットが通過すべきノードの順序がホップバイホップで含まれている．ラベルスタック内で表現されるノードはルータだけでなく，SF も含まれるため，そのラベルスタックに基づいてパケットを転送する事で SFC を実現できる．このアプローチもまた，基礎となるネットワークトポロジを変更せずに SFC を実現するために必要な，最短経路によらないパケット転送を達成する．

Segment Routing (SR)，特に Segment Routing over IPv6 (SRv6) もまた，SFC を実装するために使用される技術の 1 つである．SR では，リンク，ノード，サービスといったネットワーク内の各エンティティを**セグメント**として表現する．SRv6 パケットのヘッダ (SRH) には，セグメントリストと呼ばれる，そのパケットが通過すべきセグメントの順序を示したリストが含まれている．SRv6 では，セグメントを識別するための ID (SID) として，IPv6 アドレスを使用する．言い換えれば，SRv6 は IPv6 ルーティングインフラをその基盤として利用し，SRH 内で定義された順序に従って，任意のセグメントを経由してパケットを転送する．SRv6 は，SF が実行されるノードをセグメントとして表現し，SID を割り当て，任意の順序で SF を通過するようにパケットを転送することで SFC を実現する．

SRv6 では，SF を SID で表し，セグメントリストに基づいて適切にパケットを転送をすることで，SRv6 を基盤とした SFC ネットワークを実現できる．しかし，SRv6 レイヤよりも上位にある SF の振る舞いと，基盤となる IPv6 ルーティングインフラをどのように統合するかは明確でない．例えば，IPv4 パケットのネットワークアドレストランスレータ (NAT) を SRv6 ネットワーク内の SF として考慮する場合を考える．SRv6 ネットワーク内において，IPv4 パケットは，SRv6 ヘッダ (SRH) を含む外部 IPv6 ヘッダでカプセル化される．SF で動作する NAT の実装が SRv6 に対応していない場合，SR-Proxy [19] が必要となり，ネットワーク構成や運用における複雑さが増加してしまう [20]．実装が内

部パケットへの NAT と SRv6 に則した転送動作を同時に実行できる場合，それはレイヤバイオレーションとなる．Linux には，SERA [21] という iptables を拡張したファイアウォールアプリケーションが存在する．SERA は SRH でカプセル化されたパケットについて，カプセル化された内部パケットのヘッダ情報にマッチする iptables のフィルタールールを適用できる．SRv6 での基本的な転送動作として，End と呼ばれる動作がある．SERA は iptables を拡張することで，この End 動作を処理する機能も実装されている．ただし，既に Linux カーネルには IPv6 ルーティング，及び SRv6 End 動作に関する処理が実装されている．SERA は，Linux カーネルに実装されている SRv6 機能を使わずに，独自に改良した iptables アクションによって End 動作を処理する．つまり，SERA は Linux カーネル内で統合されている IPv6 ルーティングインフラと SRv6 処理機能を使わずに，独自に拡張した iptables によって SRv6 とフィルタリングサービスとしての NF を統合している．

本論文では，既存の netfilter を内部実装に利用する SF アプリケーションの実装を変更することなく SRv6 対応 SF として扱えるようにする，End.AN.NF 提案する．End.AN.NF は Linux netfilter を NF として扱えるようにしつつ，Linux に実装されている IPv6 ルーティングインフラを活用する．End.AN.NF は受信した SRv6 内部パケットに対して，netfilter のフックポイントを透過的するように設計されている．本論文では End.AN.NF を Linux カーネル上で実装し，スループットとレイテンシを評価した．評価の結果，End.AN.NF は End.DT 4 と H.Encaps の組み合わせによる SRv6 内部パケットへの netfilter 適用と比較し て 27% 高いスループットと 3.0 マイクロ秒低いレイテンシを実現した．さらに，End.AN.NF のレイテンシは，End.DT4 と H.Encaps の組み合わせよりも 3.0 マイクロ秒低い．また，End.AN.NF のレイテンシはマイクロ秒解像度で End 動作と同じである．

1.5 本論文の目的と構成

本論文における以降の構成は次の通りである．1 章では，本論文を読むに当たって必要な技術に関する前提知識を説明し，構成及び本論文の概要を述べる．2 章では，本論文の背景と取り組む問題について説明し，提案手法の概要を述べる．3 章では，本論文の提案する新たな SRv6 End behavior である End.AN.NF についての設計や詳細な動作，及び実装について述べる．4 章では，本論文の提案する End.AN.NF の性能が実用的であるか，また Linux カーネルのメインラインに実装されている手法の組み合わせに比べてどれだけパフォーマンスが改善されたのかをスループット及びレイテンシの観点から評価する．5 章では，本論文における結論と今後の展望について述べる．

第2章　背景と問題提起

章 1 では，SFC の概念と SFC を実現するための要件を満たせるいくつかのプロトコルを説明し，近年特に注目されている SRv6 について説明した．その知識を踏まえた上で，本章では背景と本論文で解決するべき問題を提起する．

2.1　Linux と netfilter

SFC をデプロイする上で，SF を動作させる環境として Linux を選択することは有用である．章 1.1 で述べたように，SFC 環境では SF を汎用サーバや仮想マシン，コンテナの中にデプロイすることが一般的になっている．Linux 上で任意のアプリケーションを開発して，そのバイナリを動作させることは容易であり，開発に必要な基盤や情報も十分に整っている．更に Linux カーネルはコンテナメカニズムもサポートしており，SF アプリケーションをコンテナの中で動作させることも可能である．また，Linux カーネルにはパケットフォワーディング機能が実装されている．IPv4 パケットや IPv6 パケットの転送はもちろん，MPLS や VXLAN，更にはいくつかの SRv6 ビヘイビアも実装されている．SRv6 では既存のルーティングプロトコルを使って SID を経路情報として他のルータへ広告することができる．Linux には FRR [22] や gobgp [23] などのルーティングソフトウェア実装が存在し，これらを利用して SID を広告できる．

Linux カーネルには netfilter [24] と呼ばれるパケット処理フレームも実装されており，これは SF アプリケーションを開発するのに有用である．netfilter は，パケットのフィルタリングやロギング，NAT，NAPT，やその他のパケットマングリングを可能にするフレームワークである．netfilter は，iptables や nftables といったパケット処理アプリケーションの内部の実装に使われている．iptables や nftables といった，netfilter を基盤として実装されたアプリケーションのことを，本論文では netfilter-based アプリケーションと呼ぶ．netfilter-based アプリケーションの例として，iptables や nftables 以外にも，conntract-tools[25] や snort[26] といったアプリケーションも存在する．

netfilter の仕組みを使うと，任意のカーネルモジュールは Linux カーネルのネットワークスタック上で定義された特定の場所に場所にコールバック関数を設定できる．カーネルモジュールとは，Linux カーネルのソースコードそのものを書き換えず，かつマシンの再起動を必要としない Linux カーネルの機能を拡張するプログラムのことである．一般的なユーザ定義のアプリケーションはユーザ空間で動作するため，カーネル空間で特定のプログラムを実行することはできない．しかし，カーネルモジュールとして動作するプログラムは，カーネル空間で動作する．つまり，開発者はカーネルモジュールを開発すること

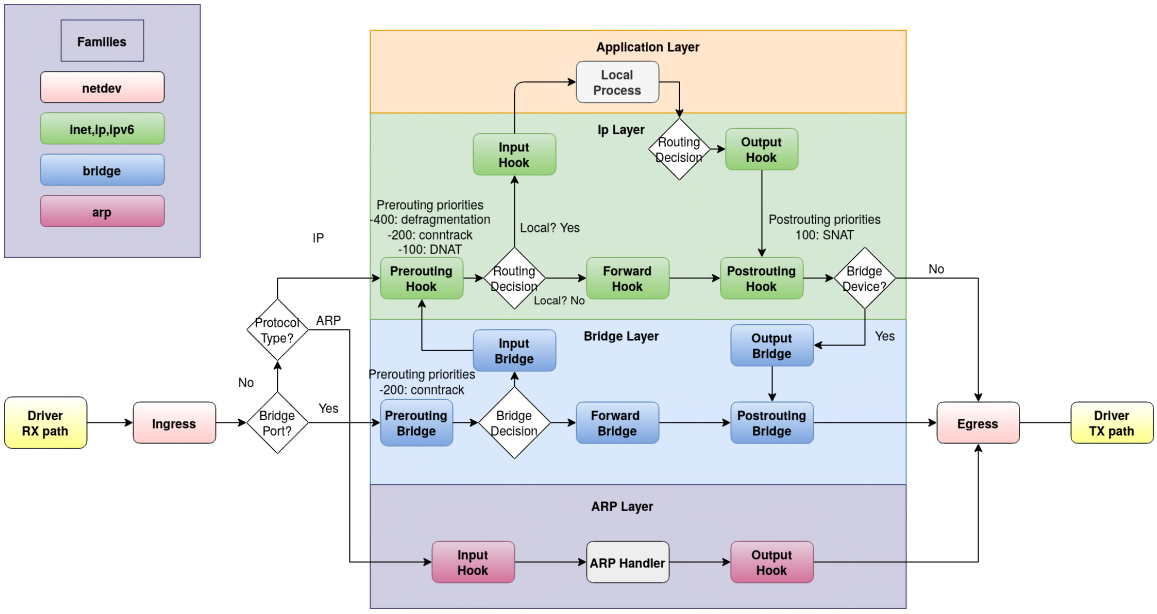

図 2.1: netfilter hook points (wiki.nftables.org より引用 [1])

で Linux カーネルに機能を追加することができる．

netfilter によってコールバック関数を設定できるポイントの一覧を，図 2.1 として示す．なお，この図は wiki.nftables.org より引用した図である．図 2.1 から読み取れるように，netfilter は Linux のネットワークスタックの様々な場所にコールバック関数を設定することができる．これらの netfilter がコールバック関数を設定できるポイントを，netfilter フックポイント，または単にフックポイント，フックという．いくつかのフックポイントは，現在処理しているパケットの宛先が自分自身かどうかによって通過するかしないかが変わる．例えば，IP レイヤに存在するフックポイントでは，パケットの宛先アドレスが自分であれば Input フックポイントを通過するが，そうでない場合は Forward フックポイントを通過する．Forward フックにのみ特定のパケットをフィルタリングするコールバック関数を登録することで，自身宛のパケットはフィルタリングをかけないが，転送するパケットにのみフィルタリングを適用する，という使い方ができる．

2.2 SRv6 と SF としての netfilter 統合手法

Linux netfilter と SRv6 を利用して SFC をデプロイすることは有用な手法であるように思えるが，実際には問題が存在する．章 2.1 で述べたように，Linux カーネルには SRv6 の機能が実装されており，netfilter は SF アプリケーションの開発に有用である．しかし，netfilter は SRH でカプセル化された内部のパケットに対して netfilter フックポイントを適用できない．これは，例えば転送するパケットの送信元アドレスを書き換える NAT 操作を SRv6 パケットの内部に適用しようとしても，SRH でカプセル化されているために通常のパケットと同じ操作では NAT を適用することができないからである．つまり，一

般的な IPv4 パケットに対して特定の操作を行うために実装された netfilter-based アプリケーションを SRv6 の内部 IPv4 パケットに対して適用する事はできない．

SRv6 に対応していない SF アプリケーションのことを，SR-unaware アプリケーションといい，対照的に SRv6 に対応している SF アプリケーションのことを SR-aware アプリケーションという．SR-unaware アプリケーションを SRv6 環境で SF として利用する方法はいくつか存在する．以降では，3 つの手法について解説する．

アプリケーションの実装を変更する手法

最も単純な方法は，アプリケーション自体の実装を変更し，SR-aware アプリケーションにすることである．SERA [21] は，Linux iptables に統合された SR-aware アプリケーションの実装である．SERA は Linux iptables を拡張し，SRH のフィールドと iptables のルールをマッチさせて，ファイアウォール用のフィルタリングルールを適用する．また，SERA は SRH でカプセル化された内部の IPv4 パケットに対してファイアウォールルールを適用する事もできる．更に SERA は，SRv6 End ビヘイビアのように，パケットを次の SID に転送する機能も持つ．しかし，SERA は iptables の拡張であるため，その他の netfilter-based アプリケーションを SR-aware にすることはできない．SERA のような手法で netfilter-based アプリケーションを SR-aware にするためには，アプリケーション毎にその実装を変える必要がある．また，SERA の採用した iptables を拡張するというデザインは，SERA に関連する SID を既存のルーティングインフラに統合することを困難にしている．iptables 内のフィルタリングルールとして利用するための SID の情報は，1.3 章で解説した layer-3 VPN の例とは異なり，既存のルーティングプロトコルを通じて広告することはできない．

SR-Proxy を利用する手法

もう 1 つの方法は，SR-Proxy と呼ばれる手法を適用することである．SR-Proxy の概念図を図 2.2 として示す．図 2.2 において，Router-B が SR-Proxy を適用するノードである．Router-B は，Router-A から受信した SRv6 パケットの SRH を一度取り外し，その状態をキャッシュしておく．Router-B は，取り外して得られた SRv6 の内部パケットを FW ノードへ送信する．FW ノードが受信するパケットは SRH でカプセル化されてない一般的な IP パケットであるため，FW ノードは受信したパケットを一般的な IP パケットとして解釈し，FW サービスを適用する．FW ノードは，FW サービスを適用したパケットを再び Router-B に送り返す．Router-B は FW サービスから受け取った非 SRv6 パケットを，キャッシュしておいた SRH で再びカプセル化する．Router-B は自身が再度カプセル化した SRv6 パケットを，次の転送先である Router-C へ送信する．

SR-Proxy を利用する方法は，汎用性が高い．SR-Proxy を実行するノードが SRH を取り外して SF ノードにパケットを転送するため，SF アプリケーションは SRH の存在を意識する必要がない．この手法であれば，iptables に限らず，あらゆる netfilter-based アプリケーションを SFC 中の SF として扱うことができる．

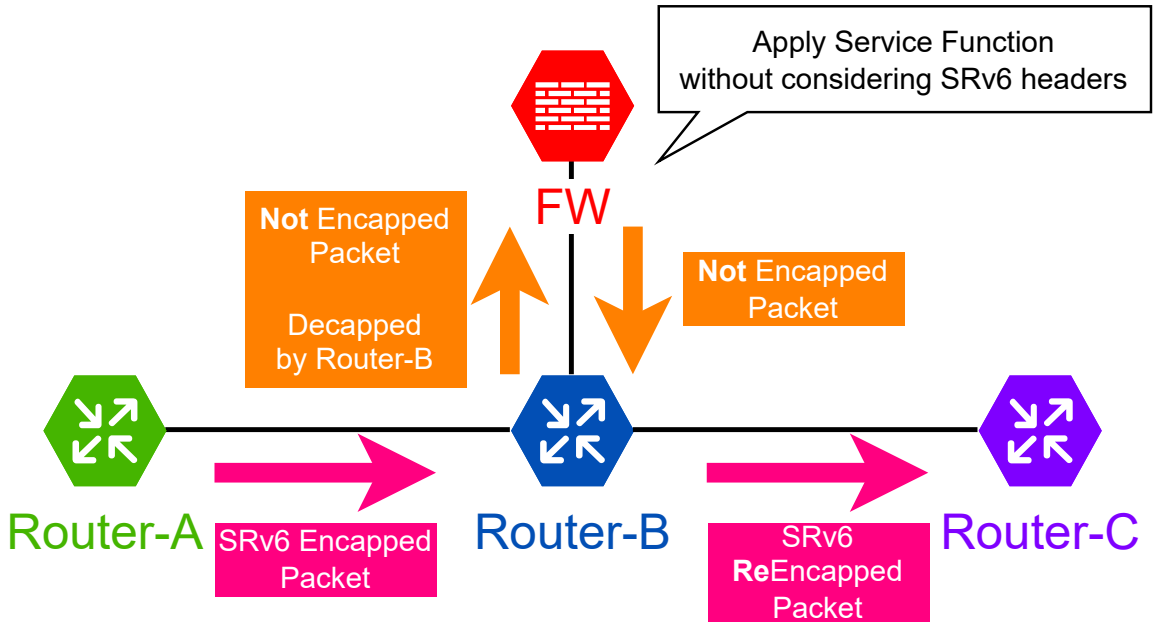

図 2.2: SR-proxy Architecture

また，SR-Proxy は SRv6 のビヘイビアとしても提案されており [27]，End.AS や End.AD や End.AM と呼ばれるビヘイビアがそれに該当する．これらは SRv6 ビヘイビアであるため，それら SID は IPv6 アドレスとして表現され，既存のルーティングプロトコルを使ってそれらの SID を広告することができる．2024 年 1 月現在，End.AD や End.AM は Linux カーネルのメインライン上では実装されておらず，これらの実装はワークインプログレス状態である．Linux で動作する SR-Proxy として，SRv6 ビヘイビア以外の実装も提案されている [28, 29, 30]．

しかし，SR-Proxy を利用する方法には，SERA のような SR-aware アプリケーションには存在しないオーバーヘッドが存在する．それは，SR-Proxy が一度 SRv6 パケットから SRH をデカプセル化する際に生じるオーバヘッド，一度外部ノードに転送し，SF 適用後に再度受信するオーバーヘッド，そして受信したパケットを再度同じ SRH でカプセル化するオーバーヘッドである．

また，SR-Proxy は根本的にネットワークにさらなる複雑さをもたらす．SR-Proxy は SF から返されるパケットに付加する適切な SID リストを決定する必要がある．SF から返される内部パケットは任意の宛先と送信元を持つ可能性があるため，SR-Proxy が付けるべき SID リストは内部パケットによって異なる可能性がある．つまり，SR-Proxy は適切な SID リストを決定するための独自のメカニズムを実装する必要がある．例えば，静的な SID リストをアタッチする End.AS か，プロキシの内部で状態をキャッシュする必要がある [28] ．さらに，SR-Proxy をデプロイするためにはいくつかの問題が存在する．例えば，特定の SR-Proxy タイプと共存できないサービスのタイプがあったり，サービスの有効性を検出する必要や SR-Proxy が送信する先の SF に関する SID 広告の問題など [20] が既にインターネットドラフトとして挙げらている．

2.3 問題提起

現在，Linux の持つ SRv6・IPv6 ルーティングインフラストラクチャを活用しながら Linux カーネルに実装されている netfilter という多機能なパケット処理機能 を NF として利用する手法は，確立されていない．現在提案されている手法では，Linux の持つ IPv6 ルーティングインフラストラクチャを活用した SR-aware アプリケーションをシンプルに汎用的に実現することは難しい．また，Linux カーネルには netfilter という多機能なパケット処理機能が実装されているものの，SRv6 上で netfilter を直接 NF として扱う方法も確立されていない．SRv6 は SF を SID として表すことで SFC を実現可能なアーキテクチャであるものの，SID として表現された SRv6 上のノードとしての SF と，実際のアプリケーションとしての SF を統合するの方法は自明ではない．セクション 2.2 で述べた SR-Proxy を利用する方法では，SR-Proxy を導入することで生まれるオーバヘッドや運用上の問題が指摘されている．また，SRH でカプセル化されているパケットに対して，パケットをカプセル化したまま netfilter 自体を適用する手法も確立されていない．本論文では，Linux netfilter を SRv6 を使って構築された SFC 上の SF として活用するための手法を提案する．

第3章　提案手法の設計と実装

本章では，前提となる Linux カーネルにおけるパケットフォワーディング処理，netfilter に関する知識を解説し，本論文の提案手法についての設計と実装について述べる．

3.1　提案手法

本論文では，netfilter を SR-aware SF として利用するための新しい実装として，Linux のルーティングインフラに netfilter によるパケットのフィルタリングとマングリング機能を統合した End.AN.NF を提案する．End.AN.NF は，End behavior of SR-aware Native function for NetFilter の略であり，これは SRv6 ビヘイビアの 1 つの種類である．End.AN.NF は netfilter-based アプリケーションを SR-aware にするのではなく，netfilter そのものを SR-aware にするという考えで設計されている．これにより既存の netfilter-based アプリケーションは，そのアプリケーションの実装を変更することなく，SRv6 で構築された SFC 環境で SF アプリケーションとして動作させることができる．End.AN.NF の実装は，Linux カーネルの IPv6 ルーティングスタックを活用するように設計されている．End.AN.NF の SID は IPv6 アドレスとして表現され，Linux 上では IPv6 のルーティングテーブルエントリとして扱われる．End.AN.NF を示す SID は，通常の IPv6 経路として既存のルーティングプロトコル，及びその実装を介して他のノードに透過的に広告される．End.AN.NF は，SRv6 パケットに対して，IPv6 パケットとして netfilter ルールを適用し，かつカプセル化されたインナーパケットに対しても同様に netfilter ルールを適用できる．End.AN.NF は，nftables[31] や iptables [32] などの netfilter-based アプリケーションを介して設定された，トラフィックに対する選択的なパケット破棄や NAT の適用などを SRH でカプセル化された内部のパケットに対しても適用できる．

3.2　設計

End.AN.NF は，トランジットするパケットを Linux ネットワークスタックの IPv6 パケット転送フローに既に存在する netfilter フックポイントを通過させ，かつ SRv6 レイヤに 3 つの netfilter フックポイントを持ち，異なるタイミングでパケットに netfilter ルールを適用する．図 3.1 は，トランジットパケットに適用される netfilter のフックのフローを示している．受信したあるパケットに対して End.AN.NF が動作する際，そのパケットには 2 段階の netfilter フックが適用される．1 段階目の適用では，SRH を含む外部 IPv6 ヘッダのついたカプセル化されたパケットに対して，その外部 IPv6 ヘッダをター

ゲットにして実行される．これは End.AN.NF が提供するものではなく，SRv6 パケットが IPv6 パケットとして解釈されて転送される際に適用されるものである．2 つ目の適用では，SRH を含まない，カプセル化された内部パケットの IP ヘッダをターゲットにして実行される．まず，End.AN.NF カーネルに実装した Linux の SRv6 ノードが IPv6 パケットを受信すると，そのカーネルは受信したパケットに prerouting フックを適用し，通常通り宛先 IPv6 アドレスの最長プレフィックスマッチングを行う．宛先アドレスが自身の持つルーティングテーブル上で End.AN.NF の SID として定義されていた場合，カーネルはパケットを End.AN.NF の実装に渡し，そうでない場合，カーネルは IPv6 レイヤの forward フックと postrouting フックを適用しながら，SRv6 パケットを IPv6 パケットとして解釈し，対応するネクストホップに転送する．一方，End.AN.NF は，SRH でカプセル化されたインナーパケットに対して，再度，prerouting フック，forward フック，及び postrouting フックを適用する．この 3 つの netfilter フックポイントは，図 2.1 で示されている通り，カーネルがあるパケットをトランジットする際に IP レイヤで通過するフックポイントである．End.AN.NF の段階で netfilter が適用されている間，SRH は End.AN.NF によって隠されるので，netfilter は SRH の処理を考慮する必要がない．End.AN.NF が終了すると，外部 IPv6 ヘッダの宛先アドレスは次の SID に置き換えられ，カプセル化されたパケットは Linux の IPv6 パケットフォワーディングプロセスにおける通常の転送パスに戻る．

End.AN.NF は，パケットをマーキングするために SID の `ARG` フィールドを利用する．SRv6 の仕様上，ある End ビヘイビアがその End ビヘイビア固有の用途で `ARG` を利用することが許可されている．End.AN.NF では，SID の `ARG` がマークとしてカーネル空間におけるパケットバッファに付加される．netfilter-based アプリケーションは，パケットバッファ上のマーク部分を照合することで，適用するルールを変更することができる．したがって，オペレータが，単一の End.AN.NF SID しか定義されていない場合であっても，SF アプリケーションは `ARG` に基づいてトラフィックのルールを調整することが可能である．

アルゴリズム 1 は，End.AN.NF がパケットを netfilter のフックポイントに渡す方法を示した擬似コードである．まず，End.AN.NF は，`ARG` の長さがこの End.AN.NF SID に指定されている場合，受信したパケットの宛先アドレスから `ARG` 値を抽出する．抽出された `ARG` 値は，マークとしてパケットバッファに付加される．次に，End.AN.NF はパケットバッファの先頭を外側の SRH から内側のパケットに切り替え，バッファを netfilter フックに渡す．フックにインストールされたルールが内側のパケットに適用された後，End.AN.NF は，パケットバッファの先頭を内側のパケットから外側の SRH に復元し，パケットを次のプロセスに渡す．この手順は，図 3.1 の赤い長方形で示した 3 つのフックポイント，prerouting，forward，postrouting に対してそれぞれ適用する．

Linux カーネルは，End ビヘイビアを特定の SID を宛先とするルーティングテーブルエントリとして扱う．End.AN.NF は End ビヘイビアの 1 つであるため，その SID も同様にルーティングテーブルエントリにインストールされる．図 3.2 に示すように，カーネルは他の End ビヘイビアと同様に End.AN.NF を表す SID をルーティングテーブルエントリとして扱っていることが確認できる．ルーティングソフトウェアや iproute2 を用い

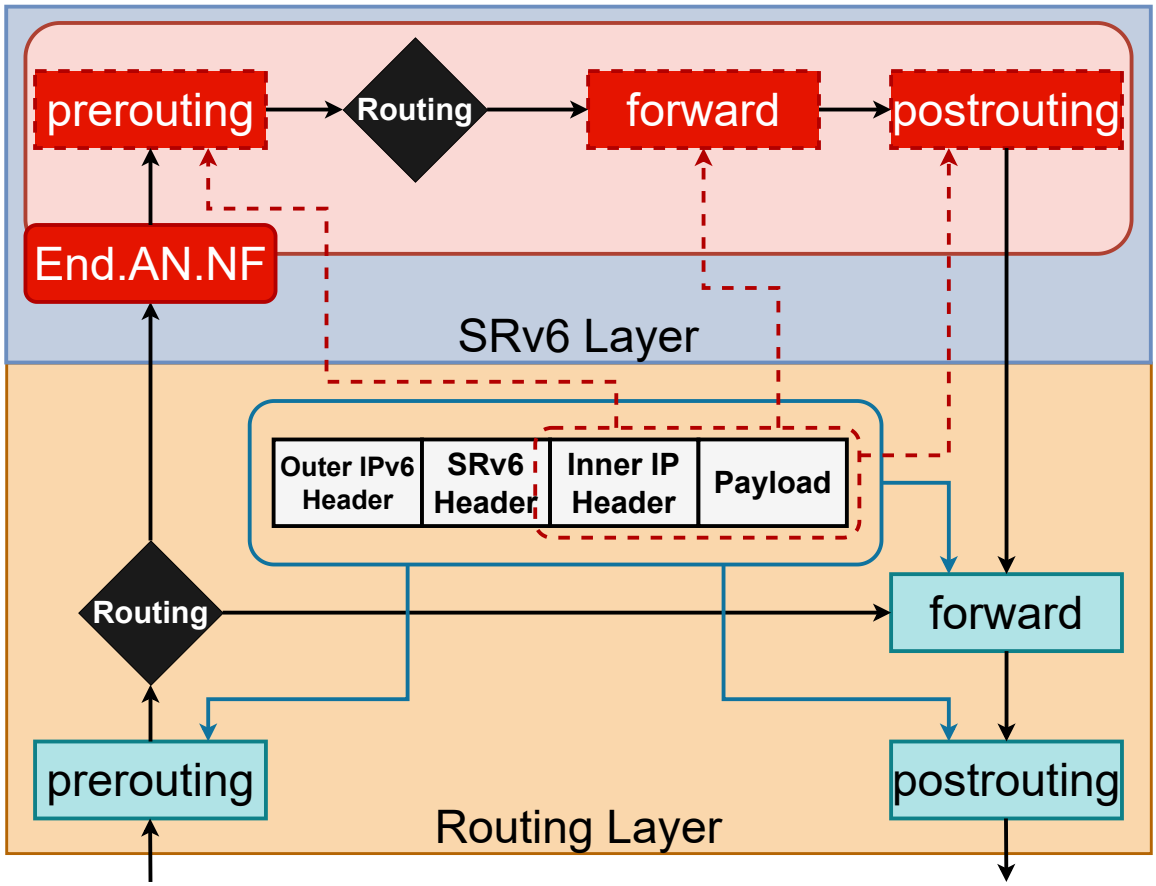

図 3.1: `End.AN.NF` applies three netfilter hook points, prerouting, forward, and postrouting, to inner packets encapsulated in SRv6.

```
$ ip -6 route | grep End
2001:db8:1::/96  encap seg6local action End.AN.NF arglen 32 dev eth0 metric 1024 pref medium
2001:db8:2::200  encap seg6local action End dev eth0 metric 1024 pref medium
2001:db8:3::300  encap seg6local action End.DX4 nh4 192.168.99.1 dev eth1 metric 1024 pref medium
```

図 3.2: The modified Linux kernel treats an `End.AN.NF` SID as an IPv6 routing table entry. We can manage the `End.AN.NF` routes with the existing tools such as iproute2.

て SID をルーティングテーブルエントリとして追加すると，従来のルーティングプロトコルを用いてカーネルのルーティングテーブルにインストールされた経路を広告することが可能となる．実際に Linux 用のソフトウェアルータ実装である FRRouting [22] を使用し，カーネル内の End.AN.NF に関連付けられた SID を BGP 経由で他のルータに IPv6 経路として広告できることを確認した．End.AN.NF のアーキテクチャは，ルーティング制御に既存のルーティングプロトコルを使用できるため，既存の SF アプリケーションとの互換性が高い．このアーキテクチャは，Linux netfilter を用いた SR-aware SF の実現方法の 1 つである．

Algorithm 1 Pseudo code of passing a packet to a netfilter hook point in `End.AN.NF`

```
1: function PassPacketToHook(packet)
2:     if the length of ARG is specified for this End.AN.NF SID then
3:         Extract the ARG value from the destination address of outer SRH
4:         Mark the ARG value on the packet buffer packet
5:     end if
6:     Switch the head of packet buffer packet from the outer SRH to the inner packet
7:     Pass packet to a netfilter hook
8:     Switch the head of packet buffer packet from the inner packet to the outer SRH
9: end function
```

3.3 実装

End.AN.NF は Linux カーネルで動作する SRv6 ビヘイビアである．End.AN.NF を実装するためには，Linux カーネルの SRv6 の実装を理解する必要がある．本セクションでは，End.AN.NF 自体の実装を解説しながら Linux カーネルにおける SRv6 の実装についても述べる．また，本論文では linux-5.15.106 を対象として End.AN.NF を実装する．ビルド時のカーネルコンフィグを付録としてとして本論文末尾に掲載する．本提案手法実装に利用した Linux フレーバーは，ubuntu のミニマムカスタムイメージである．カーネルコンフィグを編集し，動作に必要なネットワークドライバや VRF カーネルモジュールなどを有効にしている．また，本実装の動作検証はコンテナ技術を利用して仮想的なトポロジを作成して行った．コンテナを動作させるにあたり，mobyproject [33] の提供するカーネルコンフィグのチェックツール [34] を利用した．また，オーディオや GPIO サポートなど，本論文の提案手法の実装及び動作検証，計測に必要のない項目は無効化している．

3.3.1 Linux における SRv6 ビヘイビアの実装

Linux における SRv6 End ビヘイビア や End.DT4 ビヘイビアなどの実装は主に，`net/ipv6/seg6local.c` に記述されている．章 2.1 で述べたように，Linux はカーネルモジュールという仕組みを使うことで Linux のカーネルのコードそのものを書き換えないでも機能を追加実装することができる．しかし，seg6local の実装についてはカーネルモジュールとして提供されておらず，Linux カーネルのソースコード内で直接 SRv6 ビヘイビアを実装する手法が取られているため，直接 Linux カーネルのプロトコルスタックの実装を変更する必要がある．

独自の SRv6 ビヘイビアを追加するためには，まず `net/ipv6/seg6local.c` で定義されている `seg6_action_table` の末尾に要素を追加する必要がある．End.AN.NF を実装するために追加した記述を，ソースコード 3.1 として実際のコードを抜粋したものを示す．この `seg6_action_table` は `seg6_action_desc` 構造体の配列である．`seg6_action_desc` のフィールドについて，特筆するべきは `input` フィールド及び `slwt_ops` フィールドである．受信したパケットの宛先アドレスが自身の持つルーティングテーブル上で，1 つの SRv6 ビヘイビアとして表現されていた場合，そのパケットは Linux ネットワークスタッ

クの SRv6 レイヤへ到達し，パケットは SRv6 ビヘイビア毎に固有の処理の実装に渡される．input フィールドには，最初に渡される SRv6 ビヘイビア毎に固有の関数へのポインタが代入される．ソースコード 3.1 では input フィールドに input_action_end_nf 関数へのポインタが設定されている．この関数の実装については以降で解説する．slwt_ops フィールドには，SID と SRv6 ビヘイビアを経路情報としてルーティングテーブルに設定するときに呼びされる関数へのポインタが代入される．例えば End.DT4 の実装では，slwt_ops フィールドにはパケットから SRH をデカプセル化したあとにルックアップする VRF を指定するための処理が定義された関数が設定されている．End.AN.NF の場合，SID の IPv6 アドレスの下位何 bit を ARG として利用するかを指定するための処理が定義された関数である seg6_end_nf_build へのポインタが設定されている．

End.AN.NF は，SID 内部の ARG 部をパケットバッファに埋め込んだ上で，SRH でカプセル化された内部パケットに対して netfilter フックポイントを通過させる．これらの処理は input_action_end_nf 関数内で行われる．ソースコード 3.2 に，input_action_end_nf 関数内で SID 内部の ARG 部をパケットバッファに埋め込む部分の処理を示す．slwt は seg6_local_lwt 構造体へのポインタであり，これは input_action_end_nf 関数呼び出し時に引数として渡される．seg6_local_lwt 構造体には SRv6 ビヘイビアの動作に必要な様々なのフィールドが定義されいている．パケットが入ってきたインターフェースを表す数値や，End.DT4 などで使うためのルーティングテーブル ID などが含まれている．End.AN.NF の実装のために，seg6_local_lwt 構造体へ __u8 型の arg_len というフィールドを追加した．このフィールドは SID の ARG 部分の長さを示しており，このフィールドはソースコード 3.1 の slwt_ops フィールドに設定された seg6_end_nf_build 関数によって設定される．mark の計算及びパケットバッファへの埋め込みは，ARG が定義されているときにのみ行う．End.AN.NF において，ARG フィールドの利用は任意である．ARG を利用してパケットバッファにマークを付ける必要がない場合，SID を定義する際に ARG の長さを 0 とすることで ARG は無効になる．なお，ARG の長さを負の値にすることはできない．Linux では，ユーザ空間からカーネルが管理するルーティングテーブルにエントリーを追加する際，NETLINK メッセージでやり取りをする．End.AN.NF の SID を経路表に追加する際，特定のフォーマットで NETLINK メッセージを作成する．そのメッセージの中には ARG の流さを指定するフィールドが定義されており，受け取ったメッセージは seg6_end_nf_build 関数内でバリデートされ，ARG の長さが負だった場合は経路情報としてルーティングテーブルに載らない．ソースコード 3.2 では，arg_len が 0 でない，すなわち ARG が有効であるときに if 文内部の処理が実行される．計算された結果は Linux 上のパケットバッファを示す sk_buff 構造体の mark フィールドに設定される．

input_action_end_nf 関数の中で，SRH でカプセル化されたパケットを実際に netfilter フックポイントへ通過させている処理を抜粋したコードを，ソースコード 3.3 として示す．End.AN.NF は，SRv6 パケットの SRH 部分を隠蔽して netfilter にパケットを通す．ソースコード 3.3 の中で，SRH を隠蔽する，という処理は skb_pull 関数と skb_reset_network_header 関数の呼び出しによって実現される．先に述べた通り，Linux カーネルではパケットバッファを sk_buff 構造体で管理している．Linux カーネルでパケットバッファを参照して各ネットワークレイヤでパケット転送処理を実行する際，処理

ごとに sk_buff 構造体内部の head ポインタを進める必要がある．head ポインタは，パケットバファの中で現在処理をしているポインタの位置を示すものである．例えば，Ether レイヤの処理をしているときはこのポインタは Ether フレームの先頭を指す．Ether レイヤの処理が終わったあとは，head ポインタの位置を次のレイヤ，カプセル化されていない一般的なパケットであれば IP レイヤへずらす．通常このように head ポインタの位置を進める場合は，skb_pull 関数を利用する．skb_pull 関数の第一引数は sk_buff 構造体へのポインタであり，第二引数はどれだけ進めるかを整数値で渡す．ソースコード 3.3 では，第二引数に offset という変数を渡している．この変数には，予め SRH の先頭から内部パケットのヘッダまでの長さを計算して代入してある．skb_pull 関数の呼び出し後は，skb_reset_network_header 関数を使用することで，IP レイヤのヘッダ位置を再度アップデートする．

ソースコード 3.3 の 9 行目から 11 行目は実際に SRH でカプセル化された内部パケットを prerouting フックポイントへ通過させている処理である．netfilter フックポイントへの通過は，NF_HOOK マクロを呼び出すことで実現できる．netfilter フックポイントを通過させた後は skb_push 関数を呼び出しており，この関数は skb_pull 関数とは対象的に指定した分 head ポインタを前に戻す関数である．skb_push 関数の呼び出し後は，skb_pull 関数呼び出し時と同様に skb_reset_network_header 関数を呼び出してヘッダの位置をもとに戻している．

ソースコード 3.3 の 19 行目，及び 21 行目では，SRv6 End ビヘイビアに対応する処理を行っている．advance_nextseg 関数は，segleft をデクリメントして宛先アドレスを新たな SID で書き換える処理を行う関数である．また，seg6_lookup_nexthop 関数では，新たな SID で書き換えた宛先アドレスに対するネクスホップを決定している．SRv6 End ビヘイビアが行う転送処理はこの大きく分けてこの 2 つであり，この転送処理は一般的なパケット転送処理とは異なる．そのため，SRH でカプセル化された内部パケットにとってのフォワード操作，netfilter の forward フックポイントを適用するタイミングには議論の余地がある．本論文では，segleft のデクリメントと新たな SID による宛先アドレスの更新，及び新たな宛先アドレスのネクストホップの決定を，SRH でカプセル化された内部パケットにとってのフォワード操作として解釈し実装する．

ソースコード 3.3 の 26 行目，及び 27 行目では，ソースコード 3.2 と同じように ARG の値をパケットバッファのマークフィールドに設定している．パケットバッファのマークフィールドは，End.AN.NF に限らず，汎用的に利用されるフィールドである．汎用的であるため，netfilter-based アプリケーションからその値を参照して処理内容を変えることができる．ただし，その反面汎用的であるがゆえに他の用途で利用されたり，値が書き換わったりすることがある．よって，ここでは forward フックポイントを通過する前に再度マークを付け直している．パケットを forward フックポイントへ通過させる処理以降は，ほとんど同じ処理でパケットを同様に postrouting フックポイントへ通過させる．

ソースコード 3.3 に示すように，SRv6 パケットに対して End.AN.NF を使って SRH でカプセル化された内部パケットを netfilter フックポイントへ通過させる処理は非常に単純である．処理の殆どがポインタの加算及び減算になるように考慮しており，オーバーヘッドがなるべく小さくなるようにしている．

ソースコード 3.1: Add definition of End.AN.NF to seg6_action_table

```
static struct seg6_action_desc seg6_action_table[] = {
  .
  .
  .
  // その他のビヘイビアの定義
  {
    .action   = SEG6_LOCAL_ACTION_END_NF,
    .attrs    = SEG6_F_ATTR(SEG6_LOCAL_NF),
    .optattrs = SEG6_F_LOCAL_COUNTERS,
    .input    = input_action_end_nf,
    .slwt_ops = {
        .build_state = seg6_end_nf_build,
    },
  },
};
```

ソースコード 3.2: Set a mark to a packet buffer

```
static int input_action_end_fw(struct sk_buff *skb,
                           struct seg6_local_lwt *slwt)
{
  .
  .
  .
  if (slwt->arg_len) {
    memcpy(&daddr_segment, &outer_header->daddr.s6_addr32[3],
       sizeof(daddr_segment));
    arg = ntohl(daddr_segment);
    mask = (1UL << slwt->arg_len) - 1;
    arg &= mask;
    skb->mark = arg;
  }
  .
  .
  .
}
```

ソースコード 3.3: Apply netfilter to SRv6 inner packet

```
1  static int input_action_end_fw(struct sk_buff *skb,
2          struct seg6_local_lwt *slwt)
3  {
4    .
5    .
6    .
7    skb_pull(skb, offset);
8    skb_reset_network_header(skb);
9    ret = NF_HOOK(NFPROTO_IPV4, NF_INET_PRE_ROUTING,
10         dev_net(skb->dev), NULL, skb, skb->dev,
11         skb_dst(skb)->dev, dummy_okfn);
12
13   skb_push(skb, offset);
14   skb_reset_network_header(skb);
15
16   if (ret != 1)
17     return ret;
18
19   advance_nextseg(srh, &ipv6_hdr(skb)->daddr);
20
21   seg6_lookup_nexthop(skb, NULL, 0);
22
23   skb_pull(skb, offset);
24   skb_reset_network_header(skb);
25
26   if (slwt->arg_len)
27     skb->mark = arg;
28   ret = NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD,
29         dev_net(skb_dst(skb)->dev), NULL, skb, skb->dev,
30         skb_dst(skb)->dev, dummy_okfn);
31   if (ret != 1) {
32     skb_push(skb, offset);
33     skb_reset_network_header(skb);
34     return ret;
35   }
36
37   if (slwt->arg_len)
38     skb->mark = arg;
39
40   ret = NF_HOOK(NFPROTO_IPV4, NF_INET_POST_ROUTING,
41         dev_net(skb->dev), NULL, skb, skb->dev,
42         skb_dst(skb)->dev, dummy_okfn);
43
44   skb_push(skb, offset);
45   skb_reset_network_header(skb);
46
47   if (ret != 1)
48     return ret;
49
50   return dst_input(skb);
51   .
52   .
53   .
54 }
```

第4章　評価

我々が実装した End.AN.NF の性能を評価するために，3 つの実験を行った．本章では，3 つの計測実験で得られた結果から，提案手法が実用上十分なスループット性能を持っているか，及び実用的なレイテンシに収まっているのかを確認するために Linux に実装されている既存のパケット転送メカニズムと比較し評価する．このうち 2 つはスループットについて，もう 1 つはレイテンシについて焦点を当てたものである．最初の実験では，パケットサイズに基づくスループットを計測し，2 つ目の実験では，netfilter-based アプリケーションにおけるフィルタールール数を増加させた際のスループットの変化を評価した．3 つ目の実験では，異なるパケット転送メカニズムに関連するレイテンシを計測した．本章では更に，計測用パケットの送信に利用したトラフィックジェネレータ，及び評価の際に考慮したレシーブサイドスケーリングについても解説する．また，計測時に netfilter-based アプリケーションとして利用した nftables についても同様に解説する．

4.1　計測の概要と予想

End.AN.NF の性能を，3 つの転送メカニズムと比較する．比較対象は，End，End.DT4 と H.Encaps の組み合わせ，及び IPv4 である．IPv4 は Linux のパケットフォワーディング性能におけるベースラインとして参照する．図 3.1 に示すように，End.AN.NF が動作する場合，受信パケットは End と比較して 2 倍の数のフックポイントを通過する．したがって，End.AN.NF の性能は End に劣ることが予想される．一方で，End.AN.NF の性能は End.DT4 と H.Encaps の組み合わせよりも高いと予想される．

SRv6 でカプセル化されたパケットに netfilter のルールを適用する場合，バニラ Linux カーネルでの実用的なアプローチは End.DT4 と H.Encaps の組み合わせである．本論文執筆現在，Linux のメインラインには章 2.2 で示したような End.AS や End.AD などの SR-Proxy は実装されていない．本論文では，End.AS や End.AD などの SR-Proxy のかわりにバニラの Linux カーネルで動作する End.DT4 と H.Encaps の組み合わせを End.AN.NF の比較対象とする．図 4.1 に，End.AN.NF と End.DT4 と H.Encaps の組み合わせの動作の違いを示す．End.AN.NF も End.DT4 と H.Encaps の組み合わせも，どちらも SRv6 パケットとして受信したパケットの内部を netfilter フックポイントへ通過させることができる．章 3.3.1 で述べたように，End.AN.NF はパケットバッファ内で IP ヘッダを指し示す部分のポインタを操作することで SRH を隠蔽し，内部パケットを netfilter フックポイントへ通過させる．対して，End.DT4 と H.Encaps の組み合わせでは，End.DT4 が SRH を一度取り外し，netfilter を適用してから再度 H.Encaps でカプセ

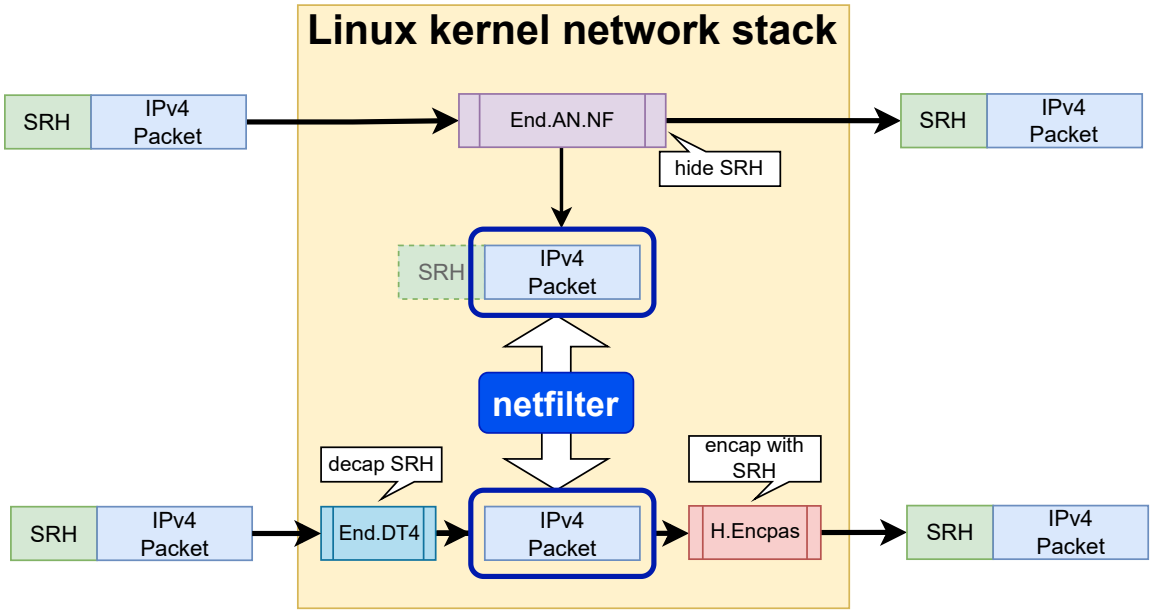

図 4.1: The difference between ”End.DT4 and H.Encaps” and End.AN.NF

ル化を行う．Linux カーネルのネットワークスタック的には，End.DT4 がデカプセル化を行うと，そのパケットは VRF で受信される．VRF で受信されたパケットは，一般的なパケットと同様に netfilter を通過する．そして，そのパケットの宛先アドレスを VRF 上でルックアップすると，経路情報として H.Encaps によって再度カプセル化されるように記述されているため，H.Encaps によってパケットはもう一度カプセル化される．つまり，End.DT4 と H.Encaps の組み合わせでは End.DT4 によるデカプセル化処理，デカプセル化されたパケットの VRF での受信，H.Encaps によるカプセル化のオーバーヘッドが存在する．したがって，このオーバーヘッドが性能の低下につながることが予想されるため，End.AN.NF の性能は End.DT4 と H.Encaps の組み合わせよりも優れていると予想できる．

3 つの実験はすべて同じ構成，同じ環境で行った．100Gbps のリンクで直結された 2 台のマシンを用意した．2 台のマシンは同一仕様で，CPU には Intel(R) Xeon(R) Silver 4310 12 コア x2，メモリは 64GB DDR4-2666，NIC には Intel E810 100Gbps を搭載している．CPU のハイパースレッディング機能は無効に設定した．1 台はトラフィック・ジェネレータとして，もう 1 台は SUT (System Under Test) として使用する．トラフィック生成マシンには Ubuntu 22.04 と TRex [35] をインストールし，テストトラフィックの生成に使用した．一方，SUT マシンには End.AN.NF を実装した Linux カーネル 5.15.106 をインストールし，End.AN.NF と，End.AN.NF の SID を設定するために独自に拡張した iproute2 コマンドを実装した．また，2 台のマシン間のリンクには 2 つの VLAN を設定し，テストトラフィックを送信するためのリンクと End.AN.NF 動作後に送信されるトラフィックが論理的に別のリンクになるようにした．VLAN は tag 付きで送信し，Linux カーネルのネットワークスタックが tag をほどく．

4.2 TRex

本論文では，スループット及びレイテンシの計測に TRex を利用した．TRex は Cisco System によって開発されたトラフィックジェネレータである．TRex は DPDK [36] というライブラリを使って開発されている．

DPDK はパケット処理をカーネル空間で行わない．Linux カーネルにはパケット転送メカニズムが実装されている．FRR などの Linux ソフトウェアルータ実装は，動作するルーティングプロトコル群がユーザ空間で動作し，NETLINK メッセージを通じて経路情報がカーネルのルーティングテーブルにインストールする．そして FRR は実際のパケット転送処理を Linux カーネルにまかせている．対して，図 4.2 に示すように DPDK では NIC で受信したパケットはカーネルをバイパスし，ユーザ空間で操作する DPDK ソフトウェアに渡される．よって，DPDK のパケット処理性能は Linux カーネルに実装されているパケット処理メカニズムの性能に依存しない．また，DPDK は CPU を独占する．DKDP アプリケーションは動作中，指定された CPU へポーリング常にを行う．これにより，コンテキストスイッチングを抑制して高速な処理を行うことができる．

TRex は柔軟なトラフィック生成が可能である．最も基本的な使い方は，元となるパケットキャプチャファイルを用意し，トラフィックごとに変更する部分を別途 yaml ファイルで定義するという方法である．この yaml ファイルには，例えば送信元 IP アドレスや宛先 IP アドレスを定義することができる TRex はこのファイルの内容に従って，元となるパケットキャプチャファイルの情報を変更し，パケットキャプチャファイルとは別の送信元 IP アドレスや宛先 IP アドレスを持つパケットを生成し，送信することができる．また，どれだけの時間，単位時間あたりにどれだけのパケットを送出するのかといったことも yaml ファイルに定義できる．

また，パケットキャプチャファイルを利用する方法以外にも，Python スクリプトを使ってパケットを生成し送出することができる．**trex_stl_lib.api** という Python ライブラリに様々な API が提供されているこのライブラリには SRv6 パケットを生成する関数も提供されており，本論文では，この Python スクリプトを使ってトラフィックを生成する手法でパケットを生成し計測した．本論文での計測実験では，IPv4 パケットを特定の SRH でカプセル化する．一部の実験時にはレシーブサイドスケーリングの仕組みを効率的に使うため，カプセル化する IPv4 パケットの送信先アドレスをインクリメントした．TRex はこのように，SRv6 のパケットの生成，及び内部パケット情報の操作も可能である．

4.3 レシーブサイドスケーリング (RSS)

レシーブサイドスケーリング (RSS) とは，マルチコアプロセッサを搭載したシステムにおいて，パケットの受信処理を複数の CPU コアに分散させる技術である．これにより，CPU コアの負荷が均等に分散され，スループットの向上が期待できる．図 4.3 に示すように，RSS では受信したパケットからハッシュを計算し，その値をもとに RX キューを分散させる．RSS の実装は NIC のドライバに依存する．ハッシュの計算アルゴリズムは

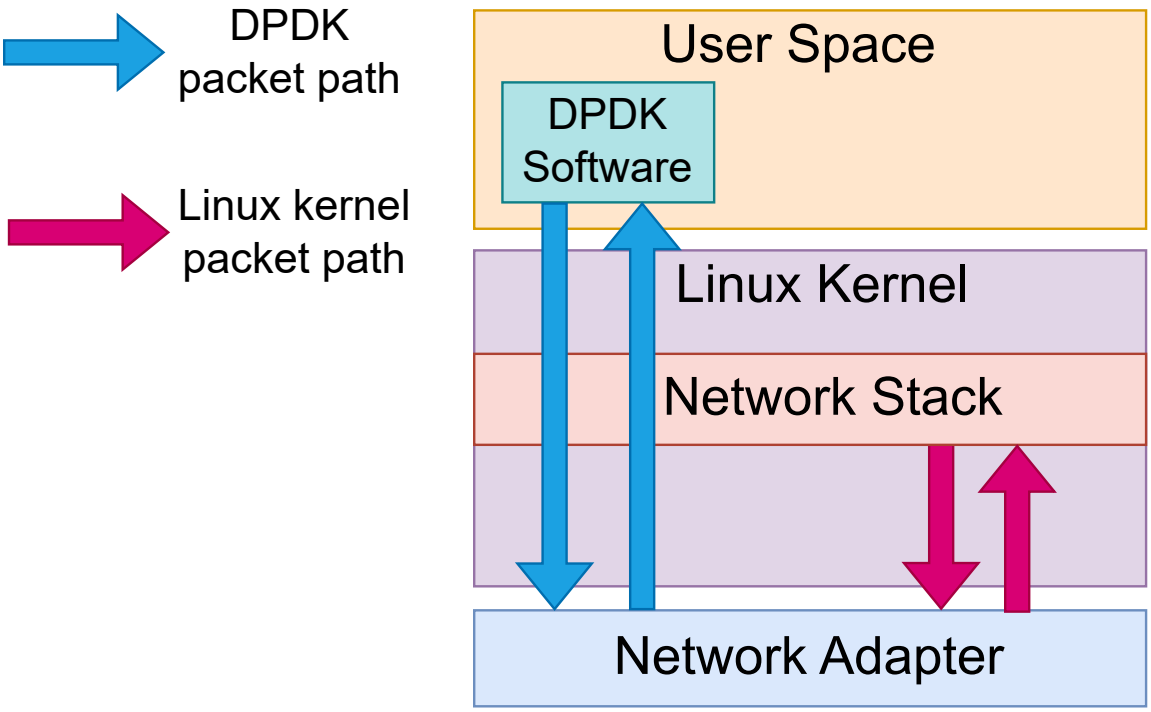

図 4.2: Abstract of DPDK

NIC のドライバの実装によって異なる上，パケットのどの部分からハッシュを計算するかも異なる．

一般的な SRv6 ネットワークではパケットを SRH でカプセル化するノードの数は限られるため，ドライバの実装が SRH の送信元及び宛先アドレスをキーにしてハッシュを計算する手法を取っている場合，ハッシュの値が偏ってしまう．本論文では，計測対象のパケットは SRv6 パケットである．SRv6 パケットは SRH でカプセル化されている．一般的な SRv6 パケットの送信元アドレスにはパケットを SRH でカプセル化するノードのループバックアドレスが割り当てられ，宛先アドレスには次の SRv6 ノードの SID が割り当てられる．つまり，SRv6 ビヘイビアを実行するノードが受け取るパケットの送信元アドレスは SRH でカプセル化するノードのループバックアドレスであり，宛先アドレスは自分自身の SID である．

本計測では，SRv6 パケットに対して RSS を有効化する必要がある際は，SRH でカプセル化された内部パケットの宛先アドレスを変更する．本計測で利用した環境では，NIC として E810 を利用した．実際にパケットキャプチャをして検証した結果，E810 では SRv6 パケットに対して RSS を適用するためには内部パケットの宛先アドレスを変更すれば良いことがわかった．SRH でカプセル化された内部パケットの宛先アドレスは，そのパケットの送信元が実際に通信するエンドノードのアドレスである．したがって，実運用上，SRH でカプセル化された内部パケットの宛先アドレスがバラバラになることは自然なシナリオである．

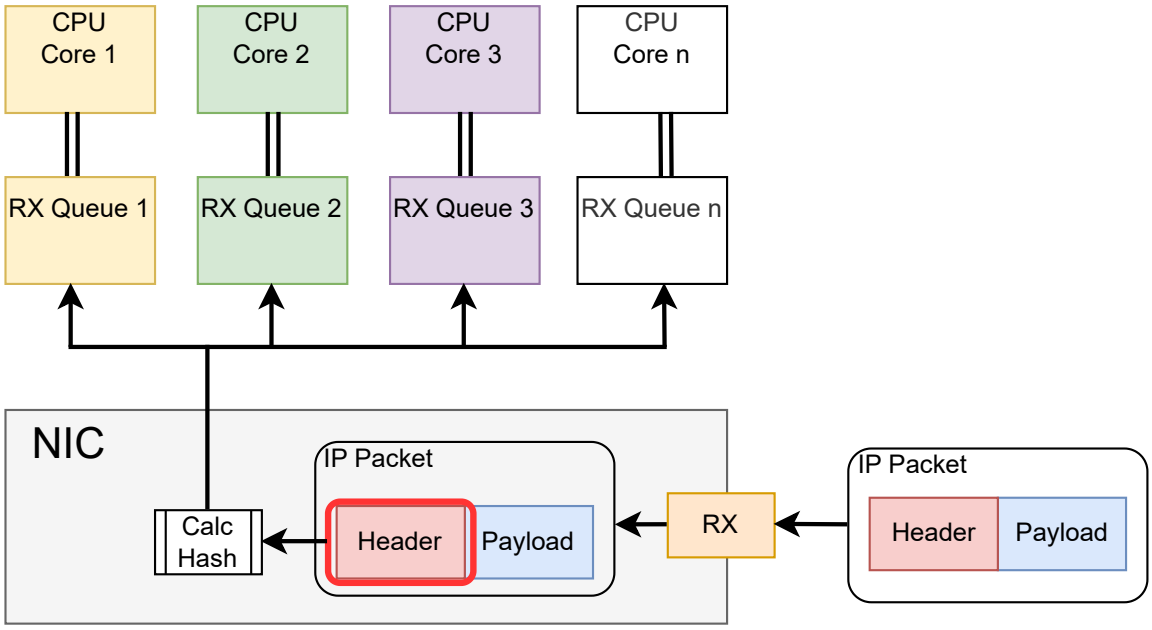

図 4.3: Abstract of RSS

4.4 パケットサイズ毎のスループット性能

End.AN.NF，End，IPv4，及び End.DT4 と H.Encaps の組み合わせについて，パケットサイズを増加させながらスループットを測定した．この実験により，各パケット転送メカニズムにおけるパケットサイズによるスループットの変化が明らかになった．この実験では，netfilter フックポイント通過時に適用されるルールはからの状態で実施した．よって，netfilter のフックポイントは通過するものの，実際にパケットにフィルタやマングルルールが適用されることはない．End.AN.NF の，End に対するスループットの低下，及び End.DT4 と H.Encaps の組み合わせに対する性能の向上を評価した．

4.4.1 計測内容

トラフィック生成マシンで TRex によって生成されたトラフィックを，最小パケット長 126 バイトから最大パケット長 1518 バイトまでパケットサイズを変化させ，SUT マシンに送信した．測定時のパケット長は次のように計算した: $l = 174n + 126$．ここで l はパケット長，n は測定回数である．$n = 0$ から $n = 10$ まで，合計 10 回の測定を行った．

最小パケット長として 126 バイトを選択した理由は，SID リストの長さが 2 である際のタグ付き VLAN を持つ UDP パケットの最小長が 126 バイトだからである．End.AN.NF は，パケットの segleft をデクリメントするため，SID リスト長は少なくとも 2 である必要がある．これは SID リストの長さが 1 の場合，segleft は 0 から始まり，End.AN.NF でデクリメントすると負の値になってしまうからである．一方，End.DT4 は，segleft が 0 であることを必要とする．End.DT4 は SRv6 ネットワークの終点で SRH をでカプセル化するビヘイビアである．つまり，End.DT4 が動作するのは SID リストによって指定さ

れた最後のノードであるため，segleft はそれ以上デクリメントできない 0 である必要がある．そこで，End.DT4 と H.Encaps の組み合わせの測定では，TRex は SID リスト長が 2 のパケットを生成し，segleft を 0 に設定した．また，レシーブサイドスケーリング (RSS) の仕組みを効果的に使用するため，TRex でパケットを生成する歳に内側の IPv4 パケットの宛先アドレスと送信元アドレスの両方をインクリメントした．IPv4 パケットの計測の際は，SRv6 パケット長に合わせて UDP ペイロードにダミーデータを埋め込み，最小パケット長が 126 バイトから始まるようにした．End.DT4 と H.Encaps の組み合わせの計測と同様，RSS を効果的に活用するため，パケット生成時に宛先アドレスと送信元アドレスをインクリメントした．最大パケット長については，タグ付き VLAN ヘッダを含むイーサフレームの最大サイズが 1518 バイトであることから，今回の測定ではパケットサイズの上限を 1518 バイトに設定した．

4.4.2 評価

図 4.4 に，この実験の結果を示す．End.AN.NF のスループットは，すべてのパケット長において End と比較して 6% 以上の低下は見られない．パケット長が 1518 バイトのとき，End.AN.NF は End と比べた際のスループットの低下が最も少なく，その低下は約 1.7% である．対して，パケット長が 478 バイトのとき，End と比較した際の End.AN.NF のスループットの低下は最も大きく，その低下は 約 5.6% である．パケット長とスループットには相関がなく，大きな変動が見られた．このスループットの低下は，End.AN.NF のパケットが End のパケットに比べて 2 倍の netfilter のフックポイントを通過することが原因として挙げられる．ただし，そのスループット低下のレベルは許容範囲内に留まっている．

End.AN.NF のスループットを End.DT4 と H.Encaps の組み合わせと比較した場合，End.AN.NF は予想通り，パケット長に関係なく一貫して優れた性能を示している．具体的には，End.AN.NF は End.DT4 と H.Encaps の組み合わせよりも，最大で 26.7% 高いスループットを達成している．グラフから，End.AN.NF と End.DT4 と H.Encaps の組み合わせとの間のスループットの差はパケット長の影響を受けていることが読み取れる．短いパケットでは相対的な性能格差が大きくなり，長いパケットではその差は縮まる．パケットサイズが小さくなるにつれて，1 秒あたりのパケット転送レート (pps) は増加する．結果として，パケットサイズが小さいほど，パケット転送のオーバーヘッドが顕著になる．

4.5 netfilter にインストールされたルール毎のスループット

次に，End.AN.NF，IPv4，および End.DT4 と H.Encaps の組み合わせについて，netfilter にインストールするフィルタールールの数を変更しながらスループットを測定した．フィルタールールのインストールには，netfilter-based アプリケーションとして nftables を使用した．nftables では，ルールはチェインの集合として表現され，チェインにはベースチェインとレギュラーチェインの 2 種類がある．nftables は，他のチェインがレギュラー

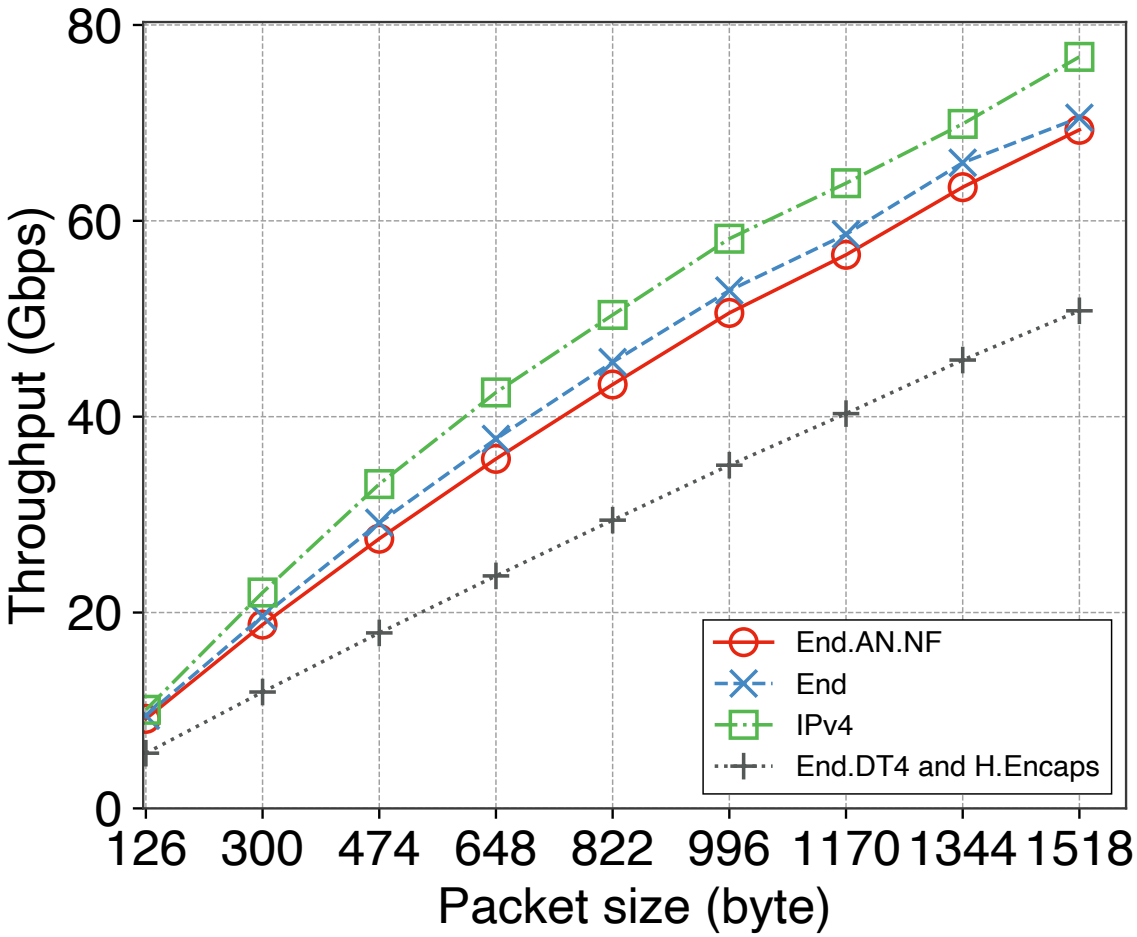

図 4.4: Throughput per SRv6 End behaviors and IPv4

チェインを参照している場合のみ，レギュラーチェインを使用する．実験では，チェインの種類ごとにカウントを増やしながらスループットを測定した．パケット転送メカニズムに関わらず，フィルタールールの追加によりスループットが低下することが予想される．この実験は，フィルタールールによる各パケット転送メカニズムのスループット低下の特徴を明らかにすることを目的とする．

4.5.1 nftables

本セクションでは，nftables について解説する．nftables 公式 wiki [37] によると，nftables とはモダンな Linux カーネル向けのパケット分類フレームワークだという．nftables を使うと，パケットのフィルタリングや NAT，NAPT やその他のパケットマングリングを適用することができる．対象は自身宛のパケットだけでなく，自信がトランジットとなる通信や自身が送出するパケットなど様々な種類のパケットに対してルールを適用できる．nftables は iptables の後継フレームワークであり，iptables の抱えるいくつかの問題点を改善したフレームワークである．例えば，nftables のルール定義文法は iptables のルール

定義文法に比べて，より構造化されている．自身の 80 番ポートに対する tcp 通信を拒否する，というフィルタールールを定義する設定ファイルを，nftables と iptables それぞれについて確認する．nftables 向けの定義ファイルをソースコード 4.1 に，iptables 向けの定義ファイルをソースコード 4.2 に示す．

nftables では，ルールセットの塊を **table** という単位で管理する．この **table** には予約されている特定の語句や記号を除き，任意の名前で定義することができる．テーブルを定義する際は，**table** *family table_name* のフォーマットで記述する．*family* には，IPv4 パケットを対象とするテーブルには **ip** が，IPv6 パケットが対象であれば **ip6** が，両方が対象であれば **inet** が入る．対して，iptables は IPv4 と IPv6 のルールを同じ枠組みで書くことができない．iptables は IPv4 向けのユーティリティであり，IPv6 には ip6tables を利用する必要がある．

ソースコード 4.1 2 行目では，`deny8080` という名前でチェインが定義されているチェインはテーブルの中に任意に数定義することができ，そのフォーマットは **chain** *chain_name* である．ソースコード 4.1 3 行目では，ネットフィルターフックポイントに登録する際に必要な情報が定義されている．**type filter** はチェインのルールがフィルターであることを表している．**hook input priority filter** は,「フィルタールールのデフォルトの優先度で，netfilter の input フックポイントに対象のチェーンを登録する」ということを示している．図 2.1 で示したように，input フックポイントは通信のエンドポイントが自分自身である際に通過するフックポイントである．ソースコード 4.1 4 行目では，実際のルールが定義されている．通信プロトコルが tcp であり，宛先ポートが 8080 番であればドロップされる．

対して iptables では，実質的なルールの定義はソースコード 4.1 の 1 行目及び 2 行目だけである．***filter** は 対象がフィルタールールであることを示し，**-A INPUT** でルールを input フックポイントに設定することを示し，以降は宛先ポートが 8080 番の tcp 通信を拒否するように書かれている．iptables では，ルールの定義をトランザクションとして管理するため，定義しただけでは変更が反映せれず，**COMMIT** キーワードで明示的に変更の反映を示す必要がある．

nftables のルール定義は，iptables のルール定義に比べて記述量が多くなる傾向がある．一方で，nftables のルール定義は iptables のルール定義に比べて宣言的で構造化されているため，ルール数が増えたときに人間が読んで理解しやすいフォーマットになっている．2 つのルールの定義方法は違うが，現在の Linux では，iptables でルールを定義すると内部的には nftables のルールに変換される．これは nftables の方が従来の iptables よりも性能面でのアドバンテージが大きいからである．

チェインにはベースチェインとレギュラーチェインの 2 種類が存在し，nftables はひとつひとつのルールをチェインという単位で管理する．ベースチェインとはソースコード 4.3 の 10 行目から 14 行目までで定義されている **filter_rule** チェインを指す．ベースチェインを宣言すると，そのチェインに対応するルールが netfilter フックポイントに設定されるため，ソースコード 4.3 の **filter_rule** チェイン内のルールは，自身が IPv4 パケットの転送動作をする際は必ず適用される．一方で，ソースコード 4.3 中の **remote_access** チェインと **rate_limit** チェインはレギュラーチェインと呼ばれるチェインである．レギュ

ラーチェインには，ソースコード 4.3 11 行目のようなルールのタイプと netfilter フックポイントを指定する項目がない．レギュラーチェインは，定義しただけではどの netfilter フックポイントにも結び付けられず，実行されない．レギュラーチェインは他のチェインからの参照によってのみ実行される．例えば，ソースコード 4.3 の 12 行目，13 行目では，パケットバッファのマークフィールドに特定の値が値が代入されていることを条件とし，**goto** キーワードを使って指定したチェインを呼び出している．このように，レギュラーチェインは **goto** キーワードで呼び出されて初めて実行される．ソースコード 4.3 ではベースチェインからレギュラーチェインを呼び出しているが，レギュラーチェインはその処理の中で別のレギュラーチェインを呼び出すこともできる．つまり，レギュラーチェインは定義しただけでは実行されず，他のベースチェインまたはレギュラーチェインから参照されて初めて実行される種類のチェインである．

ソースコード 4.1: Definition of simple filter rule for nftables

```
1 table ip filter_sample {
2   chain deny8080 {
3     type filter hook input priority filter;
4     tcp dport 8080 drop
5   }
6 }
```

ソースコード 4.2: Definition of simple filter rule for iptables

```
1 *filter
2 -A INPUT -p tcp --dport 80 -j DROP
3 COMMIT
```

ソースコード 4.3: An example of a definition: regular chain and base chain

```
1 table ip fw01 {
2   chain remote_access {
3     tcp dport ssh drop
4     tcp dport telnet drop
5     accept
6   }
7   chain rate_limit {
8     ip protocol icmp icmp type echo-request limit rate over 1/
          second drop
9   }
10  chain filter_rule {
11    type filter hook forward priority filter;
12    meta mark 0x1111 goto remote_access
13    meta mark 0x2222 goto rate_limit
14  }
15 }
```

4.5.2 計測内容

トラフィック生成マシンで TRex が生成したトラフィックを SUT マシンに送信した．この測定では，パケット長を一貫して 126 バイトに設定した．パケット長を 126 バイトに設定した理由はセクション 4.4.1 で説明したものと同じで，SID リスト長が 2 の場合のタグ付き VLAN の UDP パケットの最小長が 126 バイトだからである．

4.5.3 評価

図 4.5 は，ベースチェインのルール毎のスループットを示している．全てのチェインルールがベースチェインのみで構成されるこれらのチェインルールは，nftables のチェインルールの定義の中でも最も性能の出ないルール定義の 1 つである．この測定では，netfilter のフォワードフックポイントにフィルタールールを設定した．netfilter は 1 つのフックポイントに複数のルールを設置できる．実験を通して，このフックポイントで適用される同一のカスケードルールの数を増加させた．すべてのパケット転送メカニズムにおいて，スループットはルール数の増加と共に低下する．ルール数が増加するにつれて，3 つのパケット転送メカニズムすべてのスループットは約 0.4 Mbps に収束することがわかった．End.AN.NF と IPv4 のスループットを比較すると，スループット低下における顕著な特性の違いは見られず，End.AN.NF は IPv4 に対して大きく劣るスループット低下特性を示さない．一貫して，End.AN.NF は End.DT4 と H.Encaps の組み合わせのスループットを上回る．しかし，このスループットの差はルール数が増えるにつれて縮小し，128 ルールではわずか 9% の差まで減少した．これは，ルール数が増加するに従ってパケット転送にけるオーバーヘッドの割合が変化したからだと考えられる．ルール数が少ないうちはパケットのカプセル化とデカプセル化にかかるオーバーヘッドが割合として大きく，ルール数が増えるに従って，netfilter の処理にかかるオーバヘッドの割合が増える．netfilter の処理にかかるオーバヘッドは End.AN.NF と End.DT4 と H.Encaps の組み合わせで同じであるため，スループットの差が縮まったと考えられる．結果として，レギュラーチェインのルール数が増加するにつれて，End.AN.NF の End.DT4 と H.Encaps の組み合わせに対する優位性は低下すると言える．

図 4.6 は，ベースチェインのルール数毎のスループットを示している．注目すべき点は，ルール数を増加させてもスループットの低下が認めれず，かつ End.AN.NF が一貫して End.DT4 と H.Encaps の組み合わせを上回っていることである．レギュラーチェインのフィルタールールは，ベースチェインで測定した際と同じ構成である．通過するパケットをすべてアクセプトするルールが定義されており，一度受け入れるルールが適用されたあとも，事前に決めた回数同じ内容のルールが適用され続ける．しかし，レギュラーチェインのみから成るこのようなルール構成では，定義されたレギュラーチェインが他のチェインから参照されていないため，実際にはルールがパケットへ適用されることはない．その結果，パケットが netfilter のフックポイントを通過する際に実際に適用されるルールの数は変わらない．

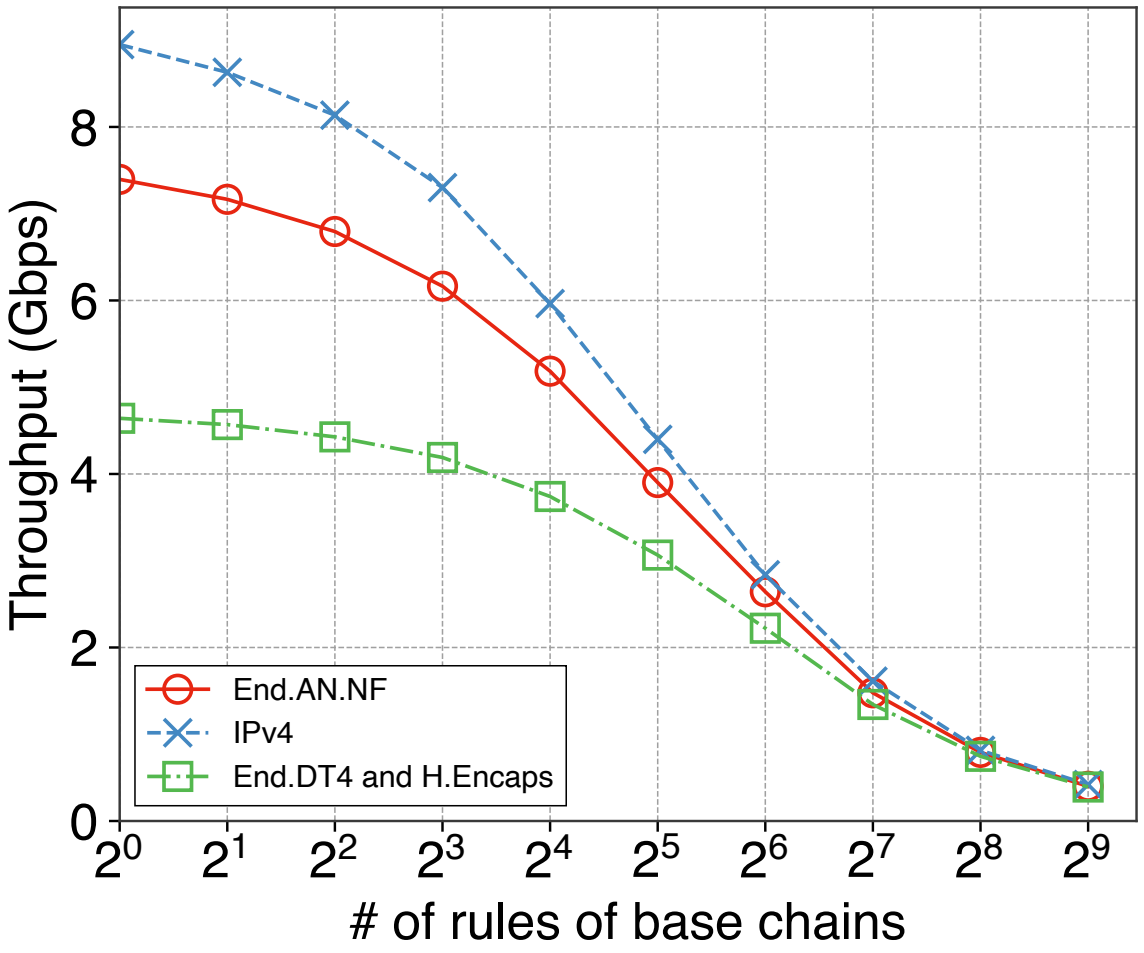

図 4.5: Throughput per number of rules of base chains

どちらの実験でも，End.AN.NF のスループットは一貫して End.DT4 と H.Encaps を組み合わせる手法を上回った．また，End.AN.NF のスループットは IPv4 には一貫して劣るものの，End.AN.NF のスループット低下特性は IPv4 と似ており，End.AN.NF のスループット性能は実用上問題ないことがわかった．

4.6 レイテンシ

スループットと同様に End.AN.NF，End，IPv4，及び End.DT4 と H.Encaps の組み合わせについて，レイテンシを測定した．この計測では特に，End.DT4 と H.Encaps の組み合わせと比較して，End.AN.NF のレイテンシがどれだけ改善されたのか評価することを目的としている．この評価では，ベースラインとして IPv4 のレイテンシを用いた．

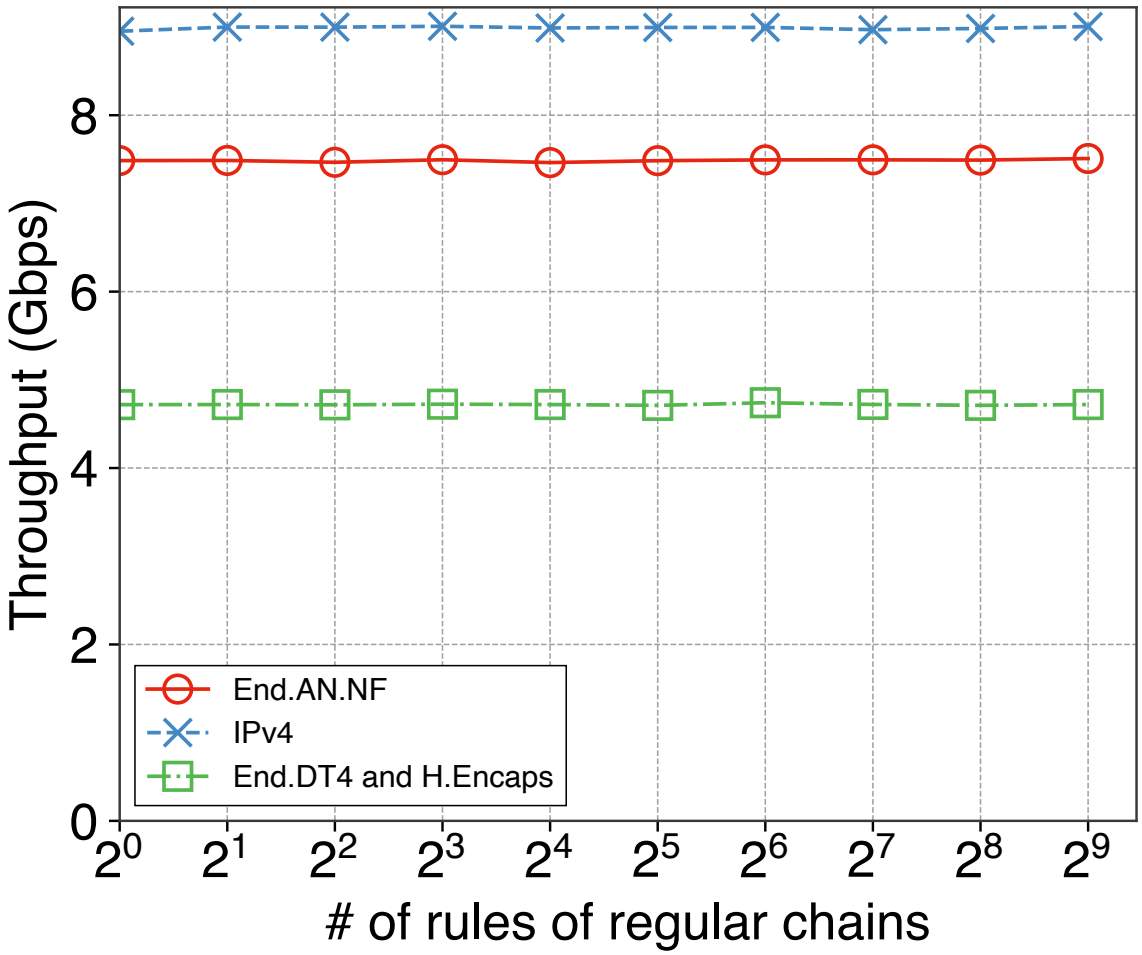

図 4.6: Throughput per number of rules of regular chains

4.6.1 計測内容

スループットと同様に，計測には TRex を使用し，パケット転送のレイテンシを測定した．TRex はパケットの送受信時間間隔をマイクロ秒単位で測ることが可能である．今回の測定は，パケット長を 142 バイトに設定した．142 バイトの内訳について，先頭 126 バイトはセクション 4.4.1 で説明した通りで，End.AN.NF がの動作要件を満たす最小のパケットとして必要だからである．追加の 16 バイトは，TRex がよるレイテンシを測定する際に利用するメタデータの埋め込みに使用される．実験中，トラフィック生成マシンは SUT に対して毎秒 10000 個のパケットを 10 秒間送信した．今回のレイテンシ測定では，RSS を無効化するために送信元アドレスと宛先アドレスを変更しなかった．なぜなら，この規模の pps で RSS を使用してパケットを処理する CPU コアを分散させてしまうと，かえってレイテンシが悪化し，余分なジッタが発生することがあるからである．

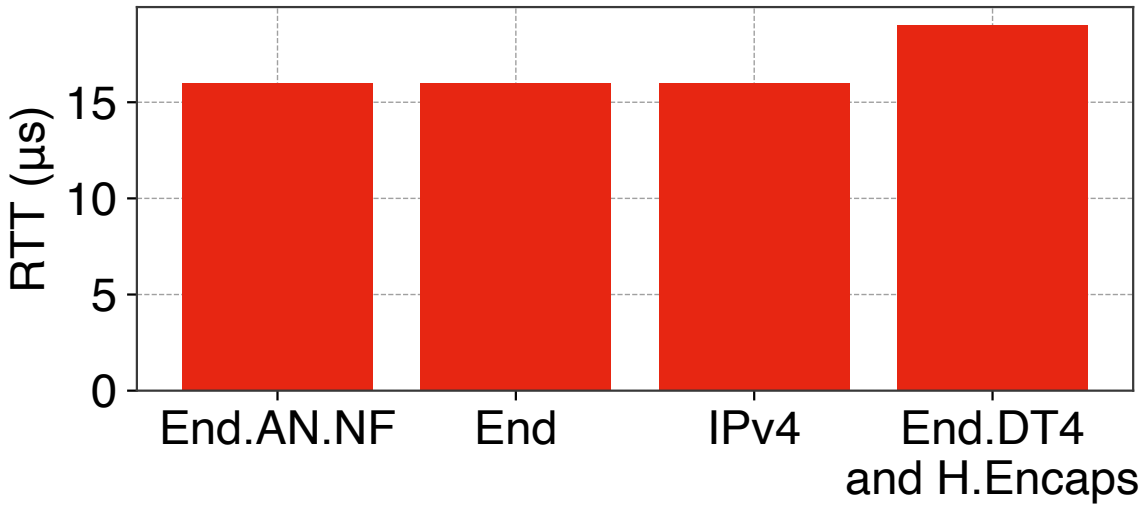

図 4.7: Latency per SRv6 End behaviors and IPv4

4.6.2 評価

計測結果を図 4.7 に示す．これらのグラフの各データポイントは，100000 回のレイテンシ測定の平均値を表している．End.AN.NF，End，IPv4 のレイテンシはどれも 16.0 マイクロ秒である．対照的に，End.DT4 と H.Encaps の組み合わせは 19.0 マイクロ秒のレイテンシである．マイクロ秒単位での測定では，End.AN.NF のレイテンシは End と IPv4 のレイテンシと一致し，End.DT4 と H.Encaps の組み合わせのレイテンシよりも約 15.8% 高速であることが分かる．よって，End.AN.NF のレイテンシはマイクロ秒単位であれば IPv4 及び End と同じであり，End.DT4 と H.Encaps の組み合わせのレイテンシよりも約 15.8% 遅延が改善された．

第5章　結論

本論文では，Linux netfilter を統合し，BGP などの既存のルーティングプロトコルとの共存を実現する新しい SRv6 End ビヘイビア，End.AN.NF を提案した．End.AN.NF は，SRv6 の内部のパケットに対して netfilter の 3 つのフックポイント prerouting，forward，postrouting を透過的に適用させることができる．netfilter のフックポイントを透過する事により，netfilter を実装に利用して作成されたアプリケーションは，その実装を変更せずに SR-aware アプリケーションとして機能させることができる．また，End.AN.NF はパケットをマークするために SID の `ARG` フィールドを活用する．このアプローチにより，netfilter を内部実装に利用した SF アプリケーションは，パケットバッファ上のマークをマッチングさせることによる動的ななルール調整が可能となる．我々は End.AN.NF を Linux カーネルに実装し，その性能を評価した．評価の結果，我々の実装は，SRv6 インナーパケットに netfilter のルールを適用する方法である End.DT4 と H.Encaps の組み合わせと比較して，27% 高いスループットと 3.0 マイクロ秒低いレイテンシを実現した．さらに，End と End.AN.NF のスループットの差は 6% 未満であり，End.AN.NF のオーバーヘッドは最も基本的な End の動作と比較して許容範囲内であることを示している．

謝辞

本論文を執筆するにあたり，ご指導いただいた慶應義塾大学教授 村井純博士，慶應義塾大学環境情報学部教授 中村修博士，同学部教授 楠本博之博士，同学部教授 高汐一紀博士，同学部教授 Rodney D. Van Meter III 博士，同学部教授 植原啓介博士，同学部教授 三次仁博士，同学部教授 中澤仁博士，同学部教授 武田圭史博士，同大学政策・メディア研究科特任教授 鈴木茂哉博士に感謝いたします．

特に植原啓介博士には rgroot のファカルティとして，日頃から研究面や運用面で指導をしていただきました．また，1 月に参加した私にとって初めての国際会議に同伴していただき，緊張している私をサポートしていただきました．感謝いたします．

私がコンピュータネットワークの分野に進む理由となった，慶應義塾大学大学院 豊田安信氏，元慶應義塾大学大学院 (現 NTT コミュニケーションズ) 深川祐太氏に感謝いたします．私は 2020 年秋学期に開講された，インターネットの設計と運用 という講義でネットワーク技術の面白さを知ることができました．豊田安信氏，深川祐太氏は TA/SA として私にネットワーク技術の面白さを伝えてくださりました．また，私を rgroot に誘ってくださったのもこのお二人でした．ありがとうございます．

東京大学准教授 中村遼博士に感謝いたします．中村遼博士には，研究面で多大な指導をしていただきました．研究ネタを一緒に考えてくださり，本論文のアイデアも中村遼博士からいただきました．また，中村遼博士に指導をしていただきながら執筆した論文は ICOIN 国際会議に採択していただくことができました．感謝いたします．

慶應義塾大学修士課程 石原匠氏に感謝いたします．石原匠氏は友人として私に接してくれながら，ときには先輩としてその背中を見せてくださりました．コロナ禍に入学した私には大学に友人が少なかったので，先輩でありながら気軽に話せる存在は大変心の支えになりました．

東京大学大学院 伊藤広記氏，元東京大学大学院 (現 LINE ヤフー株式会社) 金谷 光一郎氏に感謝いたします．伊藤広記氏，金谷 光一郎氏は，当時の私と同様にネットワーク運用未経験者として WIDE Project の vSIX ワーキンググループに参加し，共に切磋琢磨しあって頂きました．伊藤広記氏，金谷 光一郎氏は他大学の先輩でありながら，友人としても私に接してくださいました．ネットワークに入門して日が浅く右も左もわからないとき，わからないなりに共に考え，議論したことはとても良い経験になりました．

父の澤田裕司氏，母の澤田由紀氏に感謝いたします．家では口数の少ない私ですが，部屋に引きこもってパソコン作業を続けることができたのは家族のサポートあってこそでした．感謝いたします．

東京工業大学附属科学技術高等学校 13 期マイコン制御部 OB に感謝いたします．コロナ禍で大学に通えず，また新たな友人を作る機会が殆どなかった当時，同期の皆さんと毎

晩オンラインゲームに励んだことは心の支えでした．コロナ禍が明けた今でも，たまに飲みに行ったり，変わらずゲームをしたり，Twitter (X) 上で他愛もないコミュニケーションを取れることは大変嬉しいことです．本論文執筆に関しても，互いに鼓舞しあうことでモチベーションを高め合い，書き切ることができました．ありがとうございます．

全員の名前を書くことはできませんが，村井合同研，WIDE プロジェクト関係者全員に感謝いたします．私がネットワーク分野に興味を持ち，続けられたのは皆様の力あってこそでした．深く感謝申し上げます．

参考文献

[1] The Netfilter's webmasters. Netfilter hooks. https://wiki.nftables.org/wiki-nftables/index.php/Netfilter_hooks. Accessed: 2024-01-25.

[2] The Kubernetes Authors. kubernetes. https://kubernetes.io/ja/. Accessed: 2024-01-23.

[3] Abdullah Bittar, Ziqiang Wang, Amir Aghasharif, Changcheng Huang, Gauravdeep Shami, Marc Lyonnais, and Rodney Wilson. Service function chaining design & implementation using network service mesh in kubernetes. In Dhabaleswar K. Panda and Michael Sullivan, editors, *Supercomputing Frontiers*, pages 121–140, Cham, 2022. Springer International Publishing.

[4] Yakov Rekhter, Susan Hares, and Tony Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, January 2006.

[5] John Moy. OSPF Version 2. RFC 2328, April 1998.

[6] OSI IS-IS Intra-domain Routing Protocol. RFC 1142, February 1990.

[7] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, mar 2008.

[8] Paul Quinn, Uri Elzur, and Carlos Pignataro. Network Service Header (NSH). RFC 8300, January 2018.

[9] Arun Viswanathan, Eric C. Rosen, and Ross Callon. Multiprotocol Label Switching Architecture. RFC 3031, January 2001.

[10] Adrian Farrel, Stewart Bryant, and John Drake. An MPLS-Based Forwarding Plane for Service Function Chaining. RFC 8595, June 2019.

[11] Clarence Filsfils, Pablo Camarillo, John Leddy, Daniel Voyer, Satoru Matsushima, and Zhenbin Li. Segment Routing over IPv6 (SRv6) Network Programming. RFC 8986, February 2021.

[12] Gaurav Dawra, Ketan Talaulikar, Robert Raszuk, Bruno Decraene, Shunwan Zhuang, and Jorge Rabadan. BGP Overlay Services Based on Segment Routing over IPv6 (SRv6). RFC 9252, July 2022.

[13] iana: Internet Assigned Numbers Authority. IS-IS TLV Codepoints. https://www.iana.org/assignments/isis-tlv-codepoints/isis-tlv-codepoints.xhtml. Accessed: 2024-01-27.

[14] Bo Han, Vijay Gopalakrishnan, Lusheng Ji, and Seungjoon Lee. Network function virtualization: Challenges and opportunities for innovations. *IEEE Communications Magazine*, 53(2):90–97, 2015.

[15] Karamjeet Kaur, Veenu Mangat, and Krishan Kumar. A comprehensive survey of service function chain provisioning approaches in sdn and nfv architecture. *Computer Science Review*, 38:100298, 2020.

[16] Irena Trajkovska, Michail-Alexandros Kourtis, Christos Sakkas, Denis Baudinot, João Silva, Piyush Harsh, George Xylouris, Thomas Michael Bohnert, and Harilaos Koumaras. Sdn-based service function chaining mechanism and service prototype implementation in nfv scenario. *Computer Standards & Interfaces*, 54:247–265, 2017. SI: Standardization SDN&NFV.

[17] Gianluca Davoli, Walter Cerroni, Chiara Contoli, Francesco Foresta, and Franco Callegati. Implementation of service function chaining control plane through openflow. In *2017 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, pages 1–4, 2017.

[18] Deval Bhamare, Raj Jain, Mohammed Samaka, and Aiman Erbad. A survey on service function chaining. *Journal of Network and Computer Applications*, 75:138–155, 2016.

[19] Francois Clad, Xiaohu Xu, Clarence Filsfils, Daniel Bernier, Cheng Li, Bruno Decraene, Shaowen Ma, Chaitanya Yadlapalli, Wim Henderickx, and Stefano Salsano. Service Programming with Segment Routing. Internet-Draft draft-ietf-spring-sr-service-programming-08, Internet Engineering Task Force, August 2023. Work in Progress.

[20] Ryo Nakamura, Yukito Ueno, and Teppei Kamata. An Experiment of SRv6 Service Chaining at Interop Tokyo 2019 ShowNet. Internet-Draft draft-upa-srv6-service-chaining-exp-00, Internet Engineering Task Force, October 2019. Work in Progress.

[21] Ahmed Abdelsalam, Stefano Salsano, Francois Clad, Pablo Camarillo, and Clarence Filsfils. Sera: Segment routing aware firewall for service function chaining scenarios.

In *2018 IFIP Networking Conference (IFIP Networking) and Workshops*, pages 46–54, 2018.

[22] FRRouting Project, a Linux Foundation Collaborative Project. Frrouting. https://frrouting.org/. Accessed: 2023-08-23.

[23] GoBGP Community. gobgp. https://osrg.github.io/gobgp/. Accessed: 2024-01-25.

[24] The Netfilter's webmasters. netfilter: firewalling, NAT, and packet mangling for linux. https://www.netfilter.org/. Accessed: 2024-01-25.

[25] Pablo Neira Ayuso. conntract-tools: Connection tracking userspace tools for Linux. https://conntrack-tools.netfilter.org/. Accessed: 2024-01-25.

[26] Pablo Neira Ayuso. conntract-tools: Connection tracking userspace tools for Linux. https://conntrack-tools.netfilter.org/. Accessed: 2024-01-25.

[27] Clarence Filsfils, Francois Clad, Pablo Camarillo, Ahmed Abdelsalam, Stefano Salsano, Olivier Bonaventure, Jakub Horn, and Jose Liste. SRv6 interoperability report. Internet-Draft draft-filsfils-spring-srv6-interop-02, Internet Engineering Task Force, March 2019. Work in Progress.

[28] Marco Haeberle, Benjamin Steinert, Michael Weiss, and Michael Menth. A caching sfc proxy based on ebpf. In *2022 IEEE 8th International Conference on Network Softwarization (NetSoft)*, pages 171–179, 2022.

[29] Andrea Mayer, Stefano Salsano, Pier Luigi Ventre, Ahmed Abdelsalam, Luca Chiaraviglio, and Clarence Filsfils. An efficient linux kernel implementation of service function chaining for legacy vnfs based on ipv6 segment routing. In *2019 IEEE Conference on Network Softwarization (NetSoft)*, pages 333–341, 2019.

[30] Baosen Zhao, Yifang Qin, Wanghong Yang, Pengfei Fan, and Xu Zhou. Sra: Leveraging af_xdp for programmable network functions with ipv6 segment routing. In *2022 IEEE 47th Conference on Local Computer Networks (LCN)*, pages 455–462, 2022.

[31] The Netfilter's webmasters. The netfilter.org "nftables" project. https://nftables.org/projects/nftables/index.html. Accessed: 2023-09-18.

[32] The Netfilter's webmasters. The netfilter.org "iptables" project. https://nftables.org/projects/iptables/index.html. Accessed: 2023-09-18.

[33] Moby Project. moby. https://mobyproject.org/. Accessed: 2024-01-27.

[34] Moby Project. check-config.sh. https://raw.githubusercontent.com/moby/moby/master/contrib/check-config.sh. Accessed: 2024-01-27.

[35] TRex Team. Trex: Realistic traffic generator. https://trex-tgn.cisco.com/. Accessed: 2023-08-25.

[36] DPDK Project. Dpdk. https://www.dpdk.org/. Accessed: 2023-09-15.

[37] nftables.org. What is nftables? https://wiki.nftables.org/wiki-nftables/index.php/What_is_nftables%3F. Accessed: 2024-01-27.