Micro SIDs: a solution for Efficient Representation of Segment IDs in SRv6 Networks

Angelo Tulumello*, Andrea Mayer*†, Marco Bonola*†, Paolo Lungaroni*†, Carmine Scarpitta*†,
Stefano Salsano*†, Ahmed Abdelsalam§, Pablo Camarillo§, Darren Dukes§, Francoid Clad§, Clarence Filsfils§
*University of Rome Tor Vergata, †CNIT, §Cisco Systems

***Abstract*—The Segment Routing (SR) architecture is based on loose source routing. A list of instructions, called segments can be added to the packet headers, to influence the forwarding and the processing of the packets in an SR enabled network. In SRv6 (Segment Routing over IPv6 data plane) the segments are represented with IPv6 addresses, which are 16 bytes long. There are some SRv6 service scenarios that may require to carry a large number of segments in the IPv6 packet headers. Reducing the size of these overheads is useful to minimize the impact on MTU (Maximum Transfer Unit) and to enable SRv6 on legacy hardware devices with limited processing capabilities that could suffer the long headers. In this paper we present the Micro SID solution for the efficient representation of segment identifiers. With this solution, the length of the segment list can be drastically reduced.**

***Index Terms*—Segment Routing, Network Architecture, IP routing protocols**

I. Introduction

THE SRv6 (Segment Routing over IPv6) Network Programming framework [1] extends the Segment Routing architecture [2], [3]. According to [1], a *packet processing program* can be expressed with a sequence of instructions called *segments*. Each instruction is encoded in a Segment ID (SID) which is 16-byte long (128 bits, the same size of an IPv6 address). SRv6 leverages the Segment Routing Header (SRH) [4] to encode the packet processing program in the IPv6 packet headers as a *Segment List*, together with optional metadata.

In SRv6 jargon, an operation to be executed at a node is called a *behavior*. The packet processing instructions may express: i.) topological or traffic-engineering behaviours, such as “go to this node via the Best-Effort Slice” or “go to this node via the Low-Latency Slice”; ii.) fast-reroute behaviours, such as “upon the sudden loss of a link, reroute the traffic via an optimum backup path”; iii.) VPN behaviours, such as “egress the network via a specified Virtual Private Network (VPN) table of a specified Provider Edge (PE) router”. More in general, any application behaviour can be encoded in a network program, to be executed by a physical service appliance or by a softwarized component running in a virtual machine or in a container.

As discussed in [5], some application scenarios for SRv6 may require long sequences of SIDs to be carried in the SRH packet header (e.g. up to 15 SIDs). In the current SRv6 model, this requires $N * 16$ bytes to be carried in the SRH, where N is the number of SIDs in the SID list. For this reason, an open research and technological problem is to find a solution to shorten the length of the SID representation in the packet headers. In this paper we present the *Micro SID* solution [6], its implementation in three different targets and a use case showing the interoperability among them.

The Micro SID solution introduces a straightforward extension to the SRv6 network programming model: each 16-byte SID can encode a micro-program rather than a single instruction. A micro-program is composed of micro-instructions, each represented with a *Micro SID*, also called *uSID*.

In this paper we give a brief description of the SRv6 framework in Section II to explain the basic functionalities exploited in the Micro SID solution, presented in Section III. In Section IV we analyze the saving in terms of header size compared to base SRv6 obtained with the Micro SID solution and with another proposed solution called SRm6 [7]. We present the Micro SID implementation on Linux, VPP and P4 platforms in Section V and show the interoperability of the three implementations in Section VI. We evaluate the processing load performance of the Micro SID implementations in Section VII and discuss related works in Section VIII.

II. SRv6 Network Programming framework

In this section, we shortly recall the main features of SRv6 Network Programming framework, as needed to understand the rest of the paper. For further details, we refer the reader to the specification of the framework in [1] and to the tutorial on SRv6 that is available in [8].

An SRv6 SID can be partitioned in three parts and expressed as LOC:FUNCT:ARG (Locator, Function, Argument). The Locator part can be routable and used to forward a packet to a specific node, where a behavior, identified by the Function part needs to be executed. In most cases, the Argument part is not used, hence a SID can be simply decomposed in two parts LOC:FUNCT (Locator and Function). To provide an example (taken from [1]) an operator can use a /48 IPv6 network prefix for its SRv6 transport domain which include all SRv6 capable transport nodes.We refer to this prefix as *Locator Block*. Each SRv6 capable node can be assigned a different /64 IPv6 network sub-prefix inside the Locator Block, therefore up to $2^{16} = 65356$ SRv6 nodes can be supported in this specific configuration. Inside each SRv6 node, 2^{64} different SIDs can be supported. As an example (see Fig. 1, the

978-3-903176-31-7©2020 IFIP

/48 Locator Block prefix can be `fc00:1234:abcd::/48`, a specific node prefix can be `fc00:1234:abcd:N::/64`, and the SID of a behavior to be executed in the node can be `fc00:1234:abcd:0100::S`. In this case, the locator part (LOC) is represented by the leftmost 64 bits, composed by the Locator Block and by a node part N. In the example, the locator for node R_N is `fc00:1234:abcd:0N00`. The FUNCT part is represented by the rightmost 64 bits (no ARGS is considered). In the example, `0001` or `F001` are used (preceded by 12 more leading zeros in hexadecimal notation).

The regular routing protocols can be used to distribute the reachability information for the Locators associated to the SRv6 network nodes. In this way, a single routing prefix can be used to reach a given node and forward the packets towards all behaviors that can be executed by that node. To ease the interoperability, a set of "well-known" behaviors is defined in [1] (but other documents can define additional behaviors). The most important SRv6 behaviors defined in [1] are briefly described hereafter.

The simplest SRv6 behavior is the *End* behavior, which is used to enforce a topological waypoint in the path of a packet towards its final destination. In the example shown in Fig. 1, a packet coming from Site A enters the SR domain in node R_1, where it is encapsulated in an IPv6 outer packet. Starting from node R_1, the packet needs to cross R_8, then R_7, then it needs to reach R_2 where it will be decapsulated and sent to Site B. Each node R_N advertises a /64 prefix, in the example `fc00:1234:abcd:0N00::/64`. Considering node R_8, the `fc00:1234:abcd:0800::0001` SID is mapped into the *End* behavior in node R_8 reached with the `fc00:1234:abcd:0800::/64` prefix. The End behavior simply corresponds to "consuming" one SID in the SID list, therefore node R_8 will read the next SID in the SID list and will update the IPv6 destination address with the next SID. The End.X behavior is meant to cross-connect the packet towards a specific next hop. The End.T behavior is used to use a specific routing table for the the IPv6 route lookup (as needed for example to implement VPNs with per-customer routing tables). The End.DX6 behavior is used to decapsulate a packet, extracting it from the outer IPv6 packet, and to cross connect it to a specific IPv6 next hop. The End.DT6 behavior is used to decapsulate a packet and then to use a specific routing table for the IPv6 route lookup of the inner packet.

A. SRv6 Control Plane aspects

An operator is free to associate a SID (logically split into LOC:FUNCT or LOC:FUNCT:ARGS) to a given behavior in a given node. The specific values for the SIDs and in particular for the FUNCT part can be provisioned and managed by an SDN controller, and/or they can be advertised by routing protocols (OSPF, ISIS, BGP) with SRv6 specific extensions. We observe that by using an SDN based approach, the use of SRv6 specific routing protocol extensions is optional. An SRv6 network can be operated by only distributing node reachability information (regular IPv6 prefixes) in routing protocols, assuming that an SDN controller manages the association of node SRv6 behaviors to SID values.

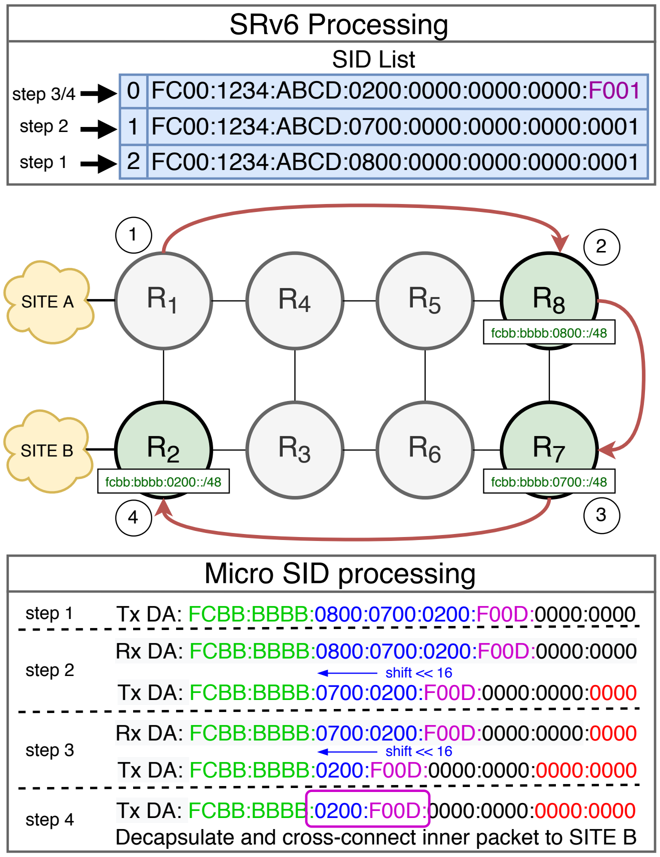
SRv6 Processing
SID List
step 3/4
0 FC00:1234:ABCD:0200:0000:0000:0000:F001
step 2
1 FC00:1234:ABCD:0700:0000:0000:0000:0001
step 1
2 FC00:1234:ABCD:0800:0000:0000:0000:0001
1
2
SITE A
R_1
R_4
R_5
R_8
fcbb:bbbb:0800::/48
SITE B
R_2
R_3
R_6
R_7
fcbb:bbbb:0200::/48
fcbb:bbbb:0700::/48
4
3
Micro SID processing
step 1
Tx DA: FCBB:BBBB:0800:0700:0200:F00D:0000:0000
step 2
Rx DA: FCBB:BBBB:0800:0700:0200:F00D:0000:0000
shift << 16
Tx DA: FCBB:BBBB:0700:0200:F00D:0000:0000:0000
step 3
Rx DA: FCBB:BBBB:0700:0200:F00D:0000:0000:0000
shift << 16
Tx DA: FCBB:BBBB:0200:F00D:0000:0000:0000:0000
step 4
Tx DA: FCBB:BBBB:0200:F00D:0000:0000:0000:0000
Decapsulate and cross-connect inner packet to SITE B

Fig. 1: Plain SID and Micro SID example

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

TABLE I: Plain SRv6 behaviors and Micro SID behaviors

III. Micro SIDs

The fundamental idea of the Micro SID solution [6] is that each 16-byte instruction (SID) of an SRv6 packet can carry a micro-program, composed of micro-instructions represented with identifiers called Micro SIDs. This approach results in a large saving of the packet overhead when multiple segments (instructions) needs to be transported in an SRv6 packet. In this work we will consider that uSIDs are represented with 2 bytes, but other choices are possible (e.g. using 3 or 4 bytes).

As described in [6], the Micro SID solution proposes to extend SRv6 Network Programming with new behaviors, called uN, uA, uDT, uDX, as described in Table I.

To introduce the reader to the basic Micro SID processing, we describe a simple use case example, based on the same reference topology of the SRv6 example, depicted in

Figure 1. In this case a /32 prefix is chosen as Locator Block for the Micro SIDs (referred to as *uSID block*). All routers in the topology are assigned a /48 prefix from this Micro SID block: `fcbb:bbbb::/32`. The ingress router R1 applies the Micro SID policy by encoding the address `fcbb:bbbb:0800:0700:0200:f00d::` into the outer IPv6 header. This results into a source routing policy that routes the packet through the path $R_8 \rightarrow R_7 \rightarrow R_2$, respectively identified by the Micro SIDs `0x0800`, `0x0700` and `0x0200` and then executes a *decap* operation. Thus, R_1 sends the packet to R_8. The packet will cross R_4 and R_5 that in this case enforce "base" IPv6 forwarding. As soon as R_8 receives the packet, it "consumes" its Micro SID identifier in the destination address: (i) the `0x0800` Micro SID is popped from the destination address; (ii) the remaining Micro SID list is shifted left by 16 bits; (iii) the End of Container identifier (`0x0000`) is inserted in the last 16 bits. The resulting IPv6 destination address is `fcbb:bbbb:0700:0200:f00d::`. Upon completion of the procedures above, the packet is transmitted to R_7 which performs an analogous set of procedures that ends with the transmission of a packet containing the Micro SID list `fcbb:bbbb:0200:f00d::` to R_2 via $R_6 \rightarrow R_3$. Since R_2 is the last SRv6 router in the path, the destination address of the packet matches the FIB entry with destination fcbb:bbbb:0200:f00d::/64. This rule includes the terminator Micro SID `f00d` which triggers the final End.DT6 behavior: the packet is decapsulated and handled by a specific IPv6 routing table.

The Micro SID solution fully leverages the SRv6 network programming solution. In particular, the data plane with the SRH dataplane encapsulation is leveraged without any change; any SID in the SID list can carry micro segments. As for the Control Plane, the SRv6 Control Plane is leveraged without any change. The mechanisms for the compression of SID identifiers are described in [9].

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

IV. Evaluation of compression savingss

This section provides a detailed analysis of the efficiency of the Micro SID compression in a realistic SRv6 deployment scenario. In particular, it considers the encapsulation size of a compressed segment lists versus an uncompressed segment list. The efficiency of the Micro SID solution is also compared with another proposed SRH compression solutions called SRm6 (Segment Routing Mapped to IPv6) [7].

We show that a mapping solution (like SRm6) does not provide better compression than what can be achieved with the SRv6 mechanism. As such, analysis of the SRm6 proposal documented in [7] is provided for comparison.

The SRm6 solution [7] defines a new routing header called *Compact Routing header* to be used to carry the list of segments instead of the SRH. More specifically, [7] defines two versions of CRH: CRH-16 and CRH-32, that respectively support Segment Identifiers (SIDs) of 16 bits (2 bytes) and 32 bits (4 bytes). The SRm6 SIDs needs to be *mapped* into IPv6 addresses, locally on each node of an SRv6 network. A "Per Path Service Instruction" can be encoded in a new Option to be included in a *Destination Option* header of the IPv6 packet.

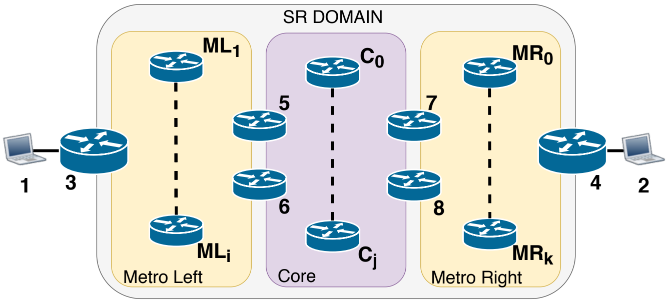

Fig. 2: Reference topology for Compression Analysis

Note that the uSID solution is fully compatible at the data plane level with the SRv6 framework, as the packet forwarding is based on IPv6 Destination Addresses and on the SRH. The SRm6 requires a data plane based on a new Routing Header and on a new Option in the Destination Option header.

A. Reference topology and scenario

Let us consider a service provider offering a VPN service with underlay optimization. The reference topology is depicted in Fig. 2. Hosts 1 and 2 are located in two different sites of a VPN customer. When host 1 sends a packet to host 2, the SR domain ingress router 3 steers it to the egress edge router 4 via an SR Policy that enforces a path through a number of underlay waypoints in Metro L (ML_1..ML_i), Core (C_1..C_j), and Metro R (MR_1..MR_k). The SR Policy ends with a SID that instructs the egress edge router 4 to decapsulate the packet and forward it towards host 2.

B. Compression Analysis

In the following, we analyze and compare the header lengths of the uSID and SRm6 with respect to the basic SRv6 header. In particular, we evaluate the "Encapsulation size Saving" i.e. the fraction of Encapsulation overhead that is saved using a compression solution with respect to the original (uncompressed) Encapsulation overhead introduced by the SRv6 solution based on full IPv6 SIDs and SRH.

According to [10], we define the Encapsulation size metric $E(SL)$ as the number of bytes required to encapsulate a packet traversing an SRv6 domain. It includes all the bytes of the "outer" IPv6 packet, from the beginning of the outer IPv6 packet (at layer 3) up to the beginning of the encapsulated packet. We note that the encapsulation size $E(SL)$ is a function of the Segment List Size SL, as each Segment in the SID List needs to be represented in the outer IPv6 packet.

The value of the the Encapsulation size metric is calculated for reduced SRv6 encapsulation as $E(SL) = 40$ bytes (IPv6

Header) if ($|SL| = 1$) or ($E(SL) = 40 + 8 + (|SL| - 1) * 16$) otherwise. Where 40 is the IPv6 header, 8 is the fixed part of SRH and 16 is the size of IPv6 address.

The SRv6 basic encapsulation is evaluated considering the reduced encapsulation policy ($H.Encap.Red$), defined in [1] section 5.1. The $H.Encap.Red$ policy encapsulates an IPv6 packet into an outer IPv6 packet with the SRH header. The first SID of the segment list is placed in the IPv6 Destination Address of the outer IPv6 packet and is not replicated in the SRH. If the SID list consists of only one SID, the entire SRH header may be omitted, resulting in a plain an IPv6 in IPv6 packet without the SRH extension header.

According to [11], we define the Encapsulation size Saving ES metric considering the Encapsulation size of the compressed solutions $E_c(SL)$ and the Encapsulation size of plain SRv6 without any compression encoding $E_p(SL)$, as follows: $ES(SL) = 1 - E_c(SL)/E_p(SL)$.

For the analysis of the Micro SID solution, the Locator Block identifies the SRv6 domain, while the the Node&Function Block represents the node identifier along with the function to be applied. The Argument block contains the metadata needed to carry out the behavior processing.

A 16-byte SRv6 instruction that contains a micro-program is called a uSID *container* instruction and has the structure shown in Fig. 1. We measure the capacity C_{uSID} of a uSID container as follows:

$$C_{uSID} = \left\lfloor \frac{(128 - B)}{NF} \right\rfloor$$

where B and NF are the lengths of the Locator and the Node&Function blocks, respectively.

Given a sequence S of uncompressed SIDs the length of the corresponding uSID sequence is evaluated as follows:

$$L_{uSID}(S) = \left\lceil \frac{|S|}{C_{uSID}} \right\rceil$$

For SRm6, SIDs of fixed size are used, of 16 or 32 bits which are carried in a new Routing Header called Compact Routing Header (CRH) [12]. The CRH is made of a fixed set of fields (i.e NextHdr, HdrLen, RoutingType, SegmentLeft, SID[0], SID[1]) for a total of 8 bytes and a variable length list of SIDs. The CRH must end on a 64-bit boundary otherwise it must be padded with zeros.

SRm6 expects headers with 16-bit or 32-bits defined as CRH-16 and CRH-32, respectively. In CRH-16 the length of headers is $E_{CRH16}(SL) = 40 + 8$ if $|SL| = 1$, otherwise $E_{CRH16}(SL) = 40 + \lceil(4 + |SL| * 2)/8\rceil * 8 + 8$.

In CRH-32 the length of headers is calculated as $E_{CRH32}(SL) = 40 + 8$ if $|SL| = 1$, otherwise $E_{CRH32}(SL) = 40 + \lceil(4 + |SL| * 4)/8\rceil * 8 + 8$.

In our comparison, the uSID solution is considered with 16-bit uSID length (the uSID Block size is 32 bit). The SRm6 is considered with both CRH-16 and CRH-32 routing headers.

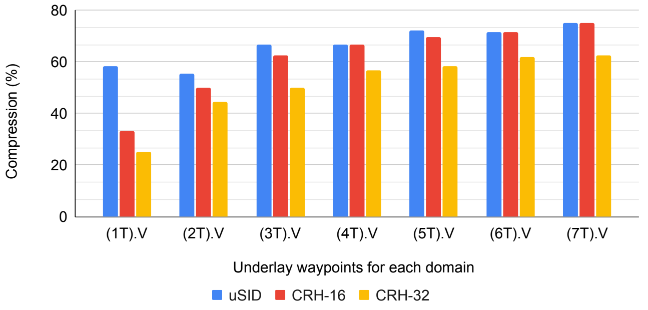

Fig. 3: Encapsulation size Saving for uSID and SRm6

Figure 3 plots the Encapsulation size Saving for the three solutions, considering the reference scenario in a range from one to seven underlay waypoints for each domain (Metro L, Core, Metro R). The Micro SID compression is significant (58%) also for a SID list of just 4 nodes (first group of bars in Figure 3), thanks to the Reduced Encapsulation in IPv6 that encodes the uSID list only in the IPv6 destination address, without adding the SRH with the SID list in the IPv6 header. Then, as the number of SIDs increases the uSID and the CRH-16 solutions align to a compression percentage around 70%.

V. Design and implementation

A. Implementation of uSIDs using P4 Language

A proof of concept implementation of uSID primitives has been realized in P4, by extending a publicly available implementation of the SRv6 framework [13]. To this end, we have developed the following extensions:

- added a new action named **usid_un**, responsible for (i) extracting the uSID of the next end router and (ii) updating the IPv6 destination address accordingly (see the Listings reported in the extended version)
- added a new Longest Prefix Match (LPM) table named **my_usid_table** responsible for the uN behavior (see listing reported in the extended version)
- modified the overall application logic (i.e.: the P4 apply block) to invoke the new processing primitives

The full P4 implementation is available in our public repository [14]. Some P4 code listings and further details are available in the extended version [15].

To support the uN behavior, the implemented P4 pipeline requires two kinds of match/action entries. The first one matches on a /48 IPv6 prefix (e.g. `fcbb:bbbb:0100::/48`) and invokes the `usid_un` action performing the shift-and-lookup primitive. The second one matches on a /64 prefix (e.g. `fcbb:bbbb:0100::/64` and triggers the SRv6 `End` behavior, i.e. decrement the SRH `segment_left` field and copy the next SID from the SRH to the IPv6 destination address.

B. Linux kernel uSID implementation

In order to add the support for uSID in the Linux kernel, we designed and implemented a patchset that extends and enhances the existent SRv6 subsystem. The proposed uSID implementation comes up with the support for the uN and uA behaviors which are, respectively, a variant of the Endpoint (End) and of the Endpoint with Cross Connect (End.X). Moreover, we have also extended the userspace `iproute2` suite [16] to support the new uSID behaviors. In particular, using the `ip` command we are able to instantiate and destroy instances of uN and uA behaviors.

All the SRv6 behaviors implemented in the Linux kernel share the same basic creation/setup function whose purpose consists of allocating the memory for the new behavior instance and parsing the supplied attributes. First, the basic creation/setup function does not allow to specify a custom callback on a per-behavior basis used for carrying out any sort of interaction with the rest of the kernel or for allocating some additional memory. Second, such basic approach does not support any optional attributes supplied by the userspace (which are required by the new uSID behaviors).

To implement the uN and uA behaviors we had to overcome the two limitations mentioned above. To this end we have: (1) extended the SRv6 implementation introducing two per-behavior callbacks which are called (if provided) when a new behavior instance is created and when it is going to be destroyed; (2) patched the SRv6 Linux kernel to support optional attributes for SRv6 behaviors without breaking any backward compatibility.

The patchsets for the Linux kernel and the `iproute2` suite are available in our project repository [17]. A more detailed explanation of the proposed uSID implementation can be found in the extended version of this paper [15].

C. uSID VPP implementation

Virtual Packet processor (VPP) is an open source virtual router [18]. It implements a high-performance forwarder that can run on commodity CPUs. VPP often runs on top of the Data Plane Development Kit (DPDK) [19] to achieve high speed I/O operations. DPDK maps directly the network interface card (NIC) into user-space bypassing the underlying Operating System kernel.

The packet processing architecture of VPP consists of graph nodes that are composed together. Each graph node performs one function of the processing stack such as IPv6 packets input (*ip6-input*), or IPv6 FIB look-up (*ip6-lookup*). The composition of the several graph nodes of VPP is decided at runtime. VPP supports most of the behaviors defined in [1].

We added a new VPP graph node (*sr-localsid-un*) to support the SRv6 uSID uN behavior. The new VPP graph node implements the shift-and-lookup functionality. When a new is uN behavior is created using VPP CLI/API, two separate FIB entries are created. The first FIB entry (e.g., `FC00:0000:0100::/48`) triggers a shift-and-lookup of the IPv6 destination address, while the second FIB entry (e.g., `FC00:0000:0100::/64`) triggers the SRH processing (implemented in the *sr-localsid* VPP graph node) by copying the next 128b SID from the SRH to the IPv6 destination address.

A received SRv6 packet may match either of the two FIB entries. Depending on which FIB entry the packet hits, it gets processed by a different VPP graph node. In this way we maintain the VPP performance and avoid instruction cache misses as all the packets that arrive to the VPP graph node must execute the same instruction, being either shift-and-lookup or SRH processing.

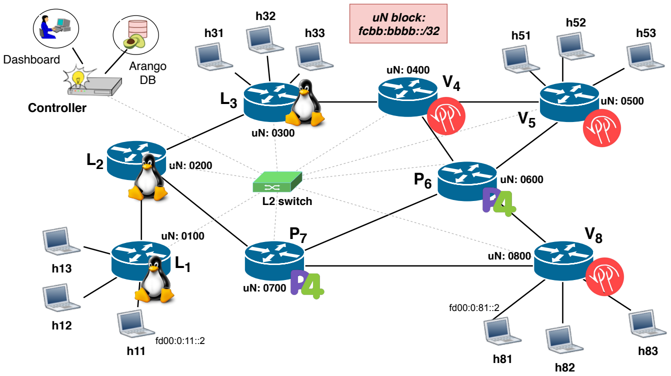

Fig. 4: uN Interoperability testbed network topology

VI. Interoperability and Testbed

A. Use case description and goals

We present a distributed use case scenario, which has two goals: (i) provide a functional assessment of the overall header compression mechanism in a meaningful application scenario; (ii) demonstrate that the uN extension can be implemented on top of different data plane frameworks and that these different implementations are inter-operable with each others. The demo of the proposed use case is similar to the SRv6 Micro SID Interoperability Demonstration presented by CISCO [20]. Differently from this one, our demo is reproducible and publicly available at the project repository [14] and includes the detailed instructions to repeat the proposed experiments.

Figure 4 shows the network topology of the proposed use case scenario, which consists of:

- 3 Linux nodes implementing the SRv6 uN functions in the kernel (L_1, L_2, L_3);
- 3 programmable data planes nodes built on top of the Vector Packet Processor platform [18] (V_4, V_5, V_8);
- 2 programmable data planes nodes built on top of the software-based P4$_{16}$ implementation bmv2 [21] (P_6, P_7);
- 1 controller responsible for managing the uN dynamic configuration of paths and host traffic to be steered;
- 12 IPv6 enabled Linux end-hosts (h11, h12, h13, etc..).

The SRv6 uN primitive set addressed by the proposed use case scenario consists of 3 functions. The first one is the `encap` function, which is responsible for encapsulating the IPv6 legacy packet that are “entering” into the SRv6 domain and specifying the uN list describing the path. This function is implemented by the edge nodes that receive packets from the transmitting end hosts. The second one is the `uN` function which is responsible for extracting the uN of the next router (as described in section III). This function is implemented

encap	**uN(un)**	**uN(End)**	**decap**
L_1, L_3	L_1, L_2, L_3	L_1, L_2, L_3	L_1, L_3
V_5, V_8	V_4, V_5, V_8	V_4, V_5, V_8	V_5, V_8
	P_7, P_6	P_7, P_6	

TABLE II: Micro SID functions and testbed nodes

both in the intermediate nodes and in edge nodes in the SRv6 path. This function operates in two different ways, referred to as `uN(un)` `uN(End)`. `uN(un)` consists in processing the active Micro SID and replacing it with the next one (through a shift operation). `uN(End)` consists in selecting the next SRv6 segment encoding a new micro-program, i.e. advancing the next SID in the SRH and copying it in the destination address of the IPv6 packet. This operation is performed by the uN behavior when there are no more Micro SIDs to be processed in the Micro SID container. Lastly the `decap` function is responsible for extracting the original IPv6 legacy packet sent by the transmitting end hosts. This function is implemented in the last (edge) nodes in the SRv6 path that are responsible for delivering the original IPv6 packet to the target end hosts.

Table II summarizes the association between the SRv6 uN functions and the nodes implementing it.

B. Testbed deployment

As the main objective of this demo is the functional assessment of the proposed header compression mechanism (the performance assessment of the proposed implementations is realized with specific standalone experiments described in Section VII), the use case scenario described in the previous section has been implemented in an emulated SW environment. In particular, we have designed and developed a virtual environment built on top of mininet [22]. The relevant mininet VM includes the 3 Micro SID implementations listed in the previous section as well as the controller and the end hosts.

Micro SID numbering. For this use case we allocated the Micro SID block `fcbb:bbbb::/32`. Each node is assigned with a Micro SID in the format `fcbb:bbbb:0X00::/44`, where X is an index bound to the node in the range $[1, .., 8]$. The use of the prefix length "/44" instead of "/48" is necessary to support the encoding of the End.DT6 directive in the least significant bit of the Micro SID (it also enables to encode 14 behaviors more). As a result, nodes with End.DT6 capability will match the "/48" rule with the first bit enabled to discriminate whether to apply uN(end) or End.DT6.

In order to solve the above mentioned issue, we implemented also an alternative solution. A special Micro SID (0xf00d) is used to support the End.DT6 and encoded at the end of the Micro SID list, e.g. `fcbb:bbbb:0X00:f00d::`. As a result, a node supporting this feature would enable the End.DT6 action when it matches its assigned Micro SID followed by `f00d:0000::`.

Further details, including the listings of the static routes configurations for Linux, P4 and VPP can be found in the extended version [15].

C. Functional assessment: control plane operations

In order to have a thorough interoperability assessment, we create multiple end host flows and associate each of them to different SRv6 uN enabled paths. For example, let us consider a bidirectional ICMP echo request/reply flow between the hosts *h11* and *h31*. For the request, the controller enforces the following path: $L_1(encap) \to L_2 \to P_7 \to P_6 \to V_5 \to V4_1(End) \to L_3(End.DT6)$. The ICMP echo reply sent by *h31* matches the same path in the reverse direction.

To express this policy from the control plane, it is just needed to trigger one command in the controller CLI that needs the following information:

- the IPv6 destination address of h31, needed to install in L_1 the path from h11 to h31;
- the IPv6 destination address of h11, to install in L_3 the path from h31 to h11;
- the list of the names of nodes to traverse (in this case `l1, l2, p7, p6, v5, v4, l3`).

The controller also implements some extended features like encoding correctly the End.DT6 behavior. As an example, it supports the corner case in which the last segment of the Micro SID list contains 6 "topological" Micro SIDs. In this case, there is no more space left in the destination address to insert the Micro SID expressing the End.DT6 behavior (0xf00d). It is also not allowed to create a new segment containing only the End.DT6 Micro SID (e.g. `fcbb:bbbb:f00d::`). Therefore, the controller automatically inserts 5 Micro SIDs in the first segment and in the last segment it inserts the Micro SID of the egress node followed by the End.DT6 Micro SID (e.g. `fcbb:bbbb:0300:f00d::`). It is worth noting that in the case of adopting the other type of uSID numbering described in Section VI-B, the entire Micro SID list would have fit inside just one IPv6 destination address, resulting in a saving of 24 bytes (8 SRH and 16 for the SID).

Other control plane features implemented for uSID include:

- creating both symmetrical (same path for both outward and return packets) and asymmetrical (different paths for outward and return packets) policies;
- dumping the list of all installed policies;
- dumping a specific policy by specifying source and destination addresses of end hosts;
- removing a policy by specifying all the parameters or by referencing the policy ID.

D. Functional assessment: data plane operations

According to the control plane configuration described in the previous subsection, the echo request sent by h11 is intercepted by L_1 that performs the `encap()` function. The original ICMP packet is encapsulated in an IPv6 header with destination address `fcbb:bbbb:0200:0700:0600:0500:0400::` expressing the first half of the path. The second half of the path is encoded in the first position of the SRH SID list with address `fcbb:bbbb:0300:f00d::`.

The encapsulated packet is then sent to L_2 which applies the uN_un function, by extracting the first Micro SID

(0200) and shifting the segment. The resulting SRv6 path is `fcbb:bbbb:0700:0600:0500:0400::`. The packet is then sent to the second uN node (P_7, identified by the Micro SID 0700). These operations are iterated until the packet reaches the last segment of the list (V_4) which applies the uN(End) function. Thus, V_4 copies the second half of the segment list in the IPv6 destination address and sends the packet to the next uN node (L_3). In L_3, acting as egress router, the packet is decapsulated and reaches the final end host h31.

For the ICMP echo reply path, the ingress node (L_1) encapsulates the packet encoding the uN list `fcbb:bbbb:0400:0500:0600:0700:0200::` in the IPv6 destination address and `fcbb:bbbb:0100:f00d::` in the SRv6 SID list. The operations applied to the reply packet are analogues to the ones applied to the request and for this reason are here omitted.

VII. PERFORMANCE EVALUATION

This section presents a performance analysis of the uN header compression mechanisms based on a set of stand alone experiments aiming at measuring the packet rate overhead introduced by the proposed extension with respect to the base SRv6 implementation.

A. Testbed deployment for the performance assessment

To evaluate both the Linux kernel and the VPP uN implementation, we have reserved two bare metal servers on the federated testbed infrastructure CloudLabs [23]. We have deployed a simple topology consisting of a traffic generator (TG) and a system under test (SUT). An instance of the TRex DPDK-based traffic generator [24] runs on the TG machine. Details of the hw configuration of the two servers are in [15].

In this simple testing scenario, we considered different bidirectional flows. Packets sent from TG are received by SUT on one network interface, processed according to the specific SRv6/Un function under measurement, and sent back to TG on the second network interface. We considered five experiments, each one with a specific combination of SRv6/uN function and packet type:

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

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

TABLE III: Linux kernel performance assessment

B. Linux kernel implementation assessment

The detailed results of the above described experiments for the Linux kernel uN implementation are reported in table III. For each experiment we performed 60 runs with a duration of 10 seconds each. Therefore, each experiment is the average of the results of the 60 runs. The throughput (848.60 kpps) measured in the experiment 5 is taken as reference to evaluate the increase or decrease in performance experienced in the other experiments. Indeed, the SRv6 End behavior does not perform any uN operation so that it allows us to find out the impact of the uSID processing with respect to the base SRv6 processing. For each experiment reported in table III, we run the performance tests to estimate the maximum throughput considering the Partial Drop Rate fixed at 0.5% (PDR@0.5%), as discussed in [25] and [26].

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

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

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

C. VPP implementation assessment

In this subsection we briefly discuss the experiment results for the VPP implementation. As expected, the packet rate measured with the VPP is one order of magnitude higher than the one obtained with the Linux kernel implementation. This is mainly due to the fact the VPP instance under measurement is built on top of DPDK[19], which compared to the plain Linux kernel network subsystem performance, provides such improved overall performance.

Indeed, for experiment 1 we measured an average packet rate of 8541.78 kpps (which, with 118 byte packets, is close to the 10 Gbps line rate of the NIC used in the testbed). For the remaining four experiments, which are all based on 158 byte packets, we always reach the line rate, i.e. 6867.59 kpps.

D. P4 implementation assessment

As described in Section V, the implementation of uN in P4 required few lines of code and as a consequence, limited resources occupation. Moreover, taking as a reference the SRv6 P4 implementation described in [13], our uN solution can even reuse the table used for SRv6 processing. This brings two advantages: (i) there is no need for adding a different table for uN processing and (ii) the P4 node remains compatible with plain SRv6. In fact, from a table occupation perspective, to support uN processing it is only needed to add two entries in the table implementing the SRv6 and uN behaviors. The P4 implementation described in this paper has not been assessed in terms of performance as it is based on a behavioral model (bmv2), meant primarily for functional assessment.

The P4 implementation presented in Section V is based on $P4_{16}$ and not compatible with existing hardware like Tofino [27] as is. In particular the $usid\_un$ action cannot be written in $P4_{14}$ as described in the Listings reported in the extended version, but must be segmented through multiple stages.

VIII. Related works

A. SRv6 protocol extensions and optimizations

A comprehensive survey of the research on SRv6 can be found in [28]. Among all the reported literature works, a considerable number is related to our work, like the ones focusing on optimizations [29][30][31][32]. A survey of the SRv6 use cases can be found in [33].

B. SRv6 header compression mechanisms

Several works addressing the compression of the SRv6 header have been proposed in literature. Indeed, within the IETF this problem is currently being addressed by several ongoing works [34][7][12][35].

The COC solution is defined in the context of the framework called "Generalized SRv6 Network Programming for SRv6 Compression" (G-SRv6) [34]. The basic idea is that in an SRv6 domain all the IPv6 SIDs can share the initial part of the address, i.e. the *Locator Block* (in the uSID solution defined in the previous section, we have called it the *uSID block*). Therefore it is possible to avoid carrying the full SID in the Segment List of the SRH. Only a node identifiers and a function (FUNCT) identifier is needed for each SID in the Segment List. In the COC/SRv6 solution, the first SID of the SRH is a regular SID, followed by a sequence of "short" identifiers called C-SIDs (Compressed-SID). At each hop, the IPv6 Destination Address (DA) will be updated keeping the Locator Block at the beginning then inserting the C-SID (node and function identifier). The final part of the address is used to encode the pointer to the currently active C-SID identifier in the C-SID list.

Note that the two proposed solutions uSID and COC have been recently combined in the same conceptual framework in [9], wherein uSID and COC are formally defined as extensions of SRv6 End and End.X flavors.

[7] and [12] propose a natively compressed version of SR mapped to IPv6 (SRm6) that inserts the SID list in an extension header of IPv6, along with a 32 bit Compressed Routing Header (CRH). Although this approach provides similar compression benefits to uSID, SRm6 needs a new control and data plane, a new ecosystem (not SRv6-native) and additional lookups at egress PE [11].

The work described in [35] proposes a mechanism to encode variable length SIDs (vSID), ranging from 1 to 128 bits, signalled by the control plane. Having SIDs of variable length increases versatility, but it comes at the cost of more complex signalling to be handled by both control and data plane.

C. Segment Routing in SDN/NFV scenarios

SRv6 has been proved to be particularly suited for SDN/NFV scenarios [36]. Abdelsalam et al. [37] explored the use of SRv6 for NFV service chaining.

A widely adopted SW based implementation of SRv6 is the one provided within the Linux kernel [38]. The performance of the Linux's SRv6 implementation has been assessed in [39].

Other relevant SW based implementations [40], [41] leverage the eBPF programmable data plane implemented in the Linux kernel to develop virtualized network functions. Another eBPF based SRv6 implementation has been exploited in [41] to realize in-network programmability use cases. Moreover, an implementation of SRv6 on P4 dataplane and ONOS controller has been presented in a tutorial[13] and has been extended with Micro SID in [42].

SR has been also exploited in SDN scenarios. Bidkar et al. [43] presented an SDN framework built upon Carrier Ethernet and augmented with SR. L. Huang et al. [44] provide a novel SR architecture based on OpenFlow that reduces the overhead of additional flow entries and label space. Dugeon et al. [45] implement and assess the SR approach with SDN based label stack optimization on top of the SDN controller OpenDaylight. Lee et al. [46] propose a routing algorithm for SDN with SR that can meet the bandwidth requirements of routing requests.

IX. Conclusions

In this paper we presented Micro SID, an extension to SRv6 that aims at reducing the protocol overhead by providing a compact representation of the segment list encoded in the IPv6 routing header (SRH). We showed an analytic demonstration of the benefit of the proposed solution and we also proved its feasibility by providing three different open source implementations that introduce negligible processing overhead with respect to the basic SRv6 approach. In addition, we presented a reproducible interoperability demonstration of the three implementations in a meaningful distributed use case.

Acknowledgment

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

References

[1] C. Filsfils et al., “SRv6 Network Programming,” Internet Engineering Task Force, Internet-Draft draft-ietf-spring-srv6-network-programming, Mar. 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-ietf-spring-srv6-network-programming

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

[3] S. Previdi et al., “Segment Routing Architecture,” IETF RFC 8402, Jul. 2018. [Online]. Available: https://tools.ietf.org/html/rfc8402/

[4] C. Filsfils, D. Dukes (ed.) et al., “IPv6 Segment Routing Header (SRH),” RFC 8754, Mar. 2020. [Online]. Available: https://tools.ietf.org/html/rfc8754

[5] W. Cheng et al., “Shorter SRv6 SID Requirements,” Internet Engineering Task Force, Internet-Draft draft-cheng-spring-shorter-srv6-sid-requirement, Jul. 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-cheng-spring-shorter-srv6-sid-requirement

[6] C. Filsfils, P. Camarillo (eds.) et al., “Network Programming extension: SRv6 uSID instruction,” Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-net-pgm-extension-srv6-usid, May 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-filsfils-spring-net-pgm-extension-srv6-usid

[7] R. B. et al., “Segment Routing Mapped To IPv6 (SRm6),” Internet Engineering Task Force, Internet-Draft draft-bonica-spring-sr-mapped-six-01, Apr. 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-bonica-spring-sr-mapped-six

[8] P. L. Ventre, S. Salsano, M. Polverini, A. Cianfrani, A. Abdelsalam, C. Filsfils, P. Camarillo, and F. Clad, “Segment routing: a comprehensive survey of research activities, standardization efforts and implementation results,” 2019.

[9] C. Filsfils (ed.) et al., “Compressed SRv6 Segment List Encoding in SRH,” Internet Engineering Task Force, Internet-Draft draft-filsfilscheng-spring-srv6-srh-comp-sl-enc, May 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-filsfilscheng-spring-srv6-srh-comp-sl-enc

[10] C. F. et al., “Analysis Framework For Extensions of SRv6 Encapsulation,” Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-analysis-fmwk-ext-srv6-encap-01, Jan. 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/filsfils-spring-analysis-fmwk-ext-srv6-encap

[11] D. Dukes, “SRv6 Network Programming Overhead Analysis,” Internet Engineering Task Force, Internet-Draft draft-dukes-spring-srv6-overhead-analysis-00, Jun. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-dukes-spring-srv6-overhead-analysis-00

[12] R. Bonica, Y. Kamite, T. Niwa, A. Alston, and L. Jalil, “The IPv6 Compact Routing Header (CRH),” Internet Engineering Task Force, Internet-Draft draft-bonica-6man-comp-rtg-hdr-22, May 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-bonica-6man-comp-rtg-hdr-22

[13] Y. T. C. Cascone, B. O’Connor. Building an srv6-enabled fabric with p4 and onos. [Online]. Available: https://github.com/opennetworkinglab/onos-p4-tutorial

[14] Micro sid interoperability testbed featuring linux kernel, vpp and p4 dataplanes. [Online]. Available: https://github.com/netgroup/usid-interop-testbed

[15] A. Tulumello et al., “Micro SIDs: a solution for Efficient Representation of Segment IDs in SRv6 Networks,” 2020. [Online]. Available: https://arxiv.org/abs/2007.12286

[16] “iproute2 website,” https://wiki.linuxfoundation.org/networking/iproute2.

[17] usid linux kernel implementation. [Online]. Available: https://netgroup.github.io/srv6-usid-linux-kernel/

[18] FD.io. Vector packet processor. [Online]. Available: https://wiki.fd.io/view/VPP

[19] DPDK. [Online]. Available: https://www.dpdk.org/

[20] SRv6 MicroSID (uSID) Interoperability Demonstration. [Online]. Available: https://www.youtube.com/watch?v=pVFkmwYIgmo

[21] P. Consortium. Behavioral model (bmv2). [Online]. Available: https://github.com/p4lang/behavioral-model

[22] Mininet: an instant virtual network on your laptop (or other pc). [Online]. Available: http://mininet.org/

[23] Cloudlab. [Online]. Available: https://www.cloudlab.us/

[24] Trex: Realistic traffic generator. [Online]. Available: https://trex-tgn.cisco.com

[25] A. Abdelsalam *et al.*, “Performance of IPv6 Segment Routing in Linux Kernel,” in *1st Workshop on Segment Routing and Service Function Chaining (SR+SFC 2018) at CNSM 2018, Rome, Italy*, 2018.

[26] A. Abdelsalam *et al.*, “SRPerf: a Performance Evaluation Framework for IPv6 Segment Routing,” *accepted to IEEE Transaction on Network and Service Management, preprint available on ArXiv: arXiv:2001.06182*, 2020.

[27] B. Networks. Product brief tofino page. [Online]. Available: https://barefootnetworks.com/products/brief-tofino/

[28] P. Ventre *et al.*, “Segment Routing: A comprehensive survey of research activities, standardization efforts and implementation results,” *arXiv preprint arXiv:1904.03471*, 2019.

[29] A. Giorgetti, P. Castoldi, F. Cugini, J. Nijhof, F. Lazzeri, and G. Bruno, “Path encoding in segment routing,” in *2015 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 2015, pp. 1–6.

[30] F. Lazzeri, G. Bruno, J. Nijhof, A. Giorgetti, and P. Castoldi, “Efficient label encoding in segment-routing enabled optical networks,” in *2015 International Conference on Optical Network Design and Modeling (ONDM)*. IEEE, 2015, pp. 34–38.

[31] A. Giorgetti, A. Sgambelluri, F. Paolucci, and P. Castoldi, “Reliable segment routing,” in *2015 7th International Workshop on Reliable Networks Design and Modeling (RNDM)*. IEEE, 2015, pp. 181–185.

[32] S. Salsano, L. Veltri, L. Davoli, P. L. Ventre, and G. Siracusano, “Pmsr—poor man’s segment routing, a minimalistic approach to segment routing and a traffic engineering use case,” in *NOMS 2016-2016 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2016, pp. 598–604.

[33] F. Duchene, M. Jadin, and O. Bonaventure, “Exploring various use cases for ipv6 segment routing,” in *Proceedings of the ACM SIGCOMM 2018 Conference on Posters and Demos*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 129–131. [Online]. Available: https://doi.org/10.1145/3234200.3234213

[34] W. Cheng, Z. Li, C. Li, F. Clad, L. Aihua, C. Xie, Y. Liu, and Shay, “Generalized SRv6 Network Programming for SRv6 Compression,” Internet Engineering Task Force, Internet-Draft draft-cl-spring-generalized-srv6-for-cmpr-, May 2020, work in Progress. [Online]. Available: https://tools.ietf.org/html/draft-cl-spring-generalized-srv6-for-cmpr

[35] B. Decraene, R. Raszuk, Z. Li, and C. Li, “SRv6 vSID: Network Programming extension for variable length SIDs,” Internet Engineering Task Force, Internet-Draft draft-decraene-spring-srv6-vlsid-03, Mar. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-decraene-spring-srv6-vlsid-03

[36] D. Lebrun, M. Jadin, F. Clad, C. Filsfils, and O. Bonaventure, “Software resolved networks: Rethinking enterprise networks with ipv6 segment routing,” in *Proceedings of the Symposium on SDN Research*, ser. SOSR ’18. New York, NY, USA: Association for Computing Machinery, 2018. [Online]. Available: https://doi.org/10.1145/3185467.3185471

[37] A. Abdelsalam, F. Clad, C. Filsfils, S. Salsano, G. Siracusano, and L. Veltri, “Implementation of virtual network function chaining through segment routing in a linux-based nfv infrastructure,” in *2017 IEEE Conference on Network Softwarization (NetSoft)*, 2017, pp. 1–5.

[38] D. Lebrun and O. Bonaventure, “Implementing ipv6 segment routing in the linux kernel,” in *Proceedings of the Applied Networking Research Workshop*, ser. ANRW ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 35–41. [Online]. Available: https://doi.org/10.1145/3106328.3106329

[39] A. Abdelsalam, P. L. Ventre, A. Mayer, S. Salsano, P. Camarillo, F. Clad, and C. Filsfils, “Performance of ipv6 segment routing in linux kernel,” in *2018 14th International Conference on Network and Service Management (CNSM)*, 2018, pp. 414–419.

[40] S. Goldshtein, “The next linux superpower: Ebpf primer,” *Dublin: USENIX Association*, 2016.

[41] M. Xhonneux, F. Duchene, and O. Bonaventure, “Leveraging ebpf for programmable network functions with ipv6 segment routing,” in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 67–72. [Online]. Available: https://doi.org/10.1145/3281411.3281426

[42] A. Abdelsalam, A. Tulumello, M. Bonola, S. Salsano, and C. Filsfils, “Pushing Network Programmability to the Limits with SRv6 uSID and P4,” in *3rd EuroP4 Workshop (EuroP4’20)*, 2020.

[43] S. Bidkar, A. Gumaste, P. Ghodasara, S. Hote, A. Kushwaha, G. Patil, S. Sonnis, R. Ambasta, B. Nayak, and P. Agrawal, "Field trial of a software defined network (sdn) using carrier ethernet and segment routing in a tier-1 provider," in *2014 IEEE Global Communications Conference*. IEEE, 2014, pp. 2166–2172.

[44] L. Huang, Q. Shen, W. Shao, and C. Xiaoyu, "Optimizing segment routing with the maximum sld constraint using openflow," *IEEE Access*, vol. 6, pp. 30 874–30 891, 2018.

[45] O. Dugeon, R. Guedrez, S. Lahoud, and G. Texier, "Demonstration of segment routing with sdn based label stack optimization," in *2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN)*. IEEE, 2017, pp. 143–145.

[46] M.-C. Lee and J.-P. Sheu, "An efficient routing algorithm based on segment routing in software-defined networking," *Computer Networks*, vol. 103, pp. 44–55, 2016.