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 source routing. Within an SR enabled network, a list of instructions called segments can be added to the packet headers to influence the forwarding and the processing of the packets. 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 from the long headers. In this paper we present the Micro SID solution for the efficient representation of segment identifiers. The proposed Micro SID solution has been implemented on three different architectures (VPP, Linux, P4) and interoperability tests have been performed. We also analyze the reduction of the header size that can be achieved with Micro SIDs and compare it with other proposals for segment list compression. Our results show that the header size can be reduced up to 75%. Finally, we mention that a fundamental asset of the proposed Micro SID solution is the full compatibility and seamless interoperability with existing SRv6 architecture.**

***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 *network 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 network 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*. A series of 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". In general, any application behaviour can be encoded in a series of SIDs, to be executed by a physical service appliance or a softwarized component running in a virtual machine or in a container.

As discussed in [5], some application scenarios for SRv6 may require long series 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. We also shortly compare this solution with other solutions [7][8][9][10][11] which are described in section IX.

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. We present the Micro SID implementation on Linux, VPP and P4 platforms in Section IV and show the interoperability of the three implementations in Section V. In Section VI we describe and assess a meaningful SDN/NFV use case built on top of the proposed MicroSID framework. In Section VII 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 [8]. We evaluate the processing load performance of the Micro SID implementations in Section VIII and discuss related works in Section IX. This paper is the extended version of a conference paper [12] published by the same authors. In particular, we extended and clarified the description of the general SRv6 Framework (Section II) and the Micro SID mechanism (Section III). In Section VI we provided a new SDN/NFV use case for practical service provisioning. We extended the original P4 implementation by adding the missing SRv6 and Micro SID behaviors. Moreover, in this extended version, we broadened the evaluation section by including latency and throughput measurements for all the implemented behaviors.

II. SRv6 Network Programming framework

In this section, we 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 [13].

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 /32 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 /48 IPv6 network sub-prefix inside the Locator Block, therefore up to $2^{16} = 65536$ SRv6 nodes can be supported in this specific configuration. Inside each SRv6 node, 2^{80} different SIDs can be supported.

Let us consider the scenario depicted in Figure 1. The locator block assigned to the SRv6 enabled network is `FCBB:BBBB::/32` and the actual topology consists of 8 SRv6 routers to which a smaller prefix included in the locator block is assigned. For example R_2 is identified by the node prefix `FCBB:BBBB:0200::/48` and in general the Nth router is identified by a node prefix in the form of `FCBB:BBBB:0N00::/48`. For sake of simplicity, each router in the topology implements only two standard SRv6 behaviors [1]:

- **End**: the Endpoint behavior ("End" for short) is the most basic behavior and is responsible for decrementing the IPv6 Hop Limit by 1, decrementing the SRH Segments Left by 1, updating IPv6 DA with Segment List[Segments Left] and finally submitting the packet to the egress IPv6 FIB lookup for transmission to the new destination.
- **End.DT6**: the End.DT6 is a variant of the End behavior: when N receives a packet destined to S and S is a local End.DT6 SID, it submits the packet to the egress IPv6 FIB lookup and transmits it the new destination. The End.DT6 SID must be the last segment in a SR Policy.

For each router, the two above mentioned behaviors are respectively associated to two SIDs in the form of `FCBB:BBBB:0N00::0001/48` and `FCBB:BBBB:0N00::F001/48`. For example, the End and the End.DT6 behaviors on R_2 are respectively associated to the SIDs `FCBB:BBBB:0200::0001/48` and `FCBB:BBBB:0200::F001/48`.
In this simple scenario SRv6 is used to enforce a source routing policy and in particular we want to use the path $R_8 \rightarrow R_7 \rightarrow R_2$ for all packets from Site A to Site B. To this end, R1 performs the following operations:

1) it creates the SID list representing the segment routing path, i.e. `FCBB:BBBB:0800::0001`, `FCBB:BBBB:0700::0001` and `FCBB:BBBB:0200::F001`;
2) it encodes the SID list above into an SRv6 header;
3) it encapsulates the original packet into an IPv6 header with destination address set to the first SID in the SID list.

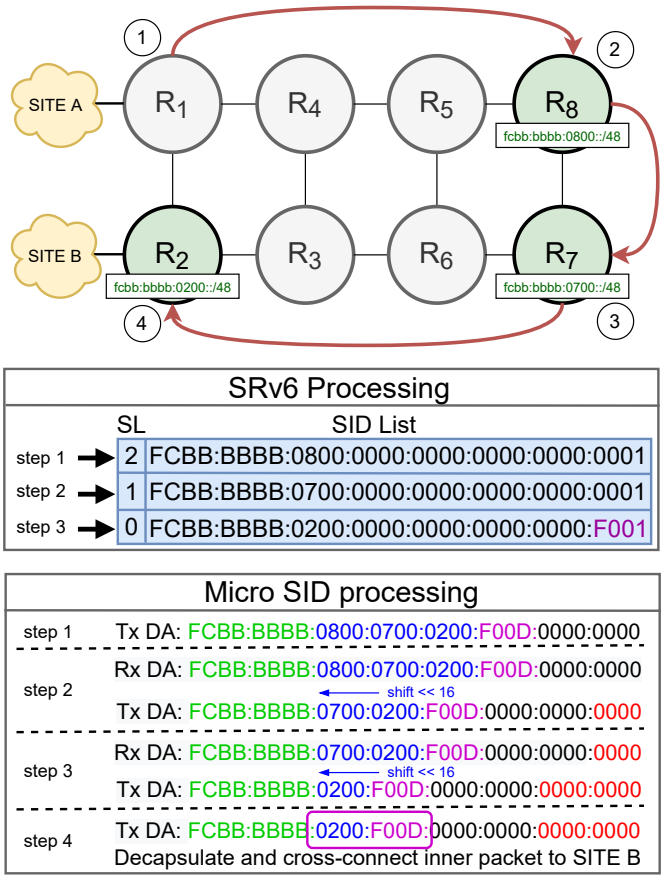

Fig. 1: Plain SID and Micro SID example

4) it embeds the SRv6 header as an IPv6 extension header
5) it sends the resulting packet to R_8 according to the IPv6 routing information exchanged in the network

The packet sent by R_1 is forwarded according to the IPv6 routing information exchanged in the network. So for example, the packet can reach R_8 by traversing R_4 and R_5, which performs standard IPv6 forwarding.
As soon as R_8 receives the packet, the End behavior is invoked. This function simply corresponds to "consuming" one SID in the SID list, therefore R_8 performs the following operations:

1) it decrements the outer IPv6 hop limit by 1;
2) it decrements the SRv6 Segments Left header field by 1;
3) it changes the outer IPv6 destination address to the second SID in the list (i.e. the one indexed by the SRv6 Segments Left field), which is `FCBB:BBBB:0700::0001`;
4) it sends the resulting packet to R_7 according to the IPv6 routing information exchanged in the network

The packet then reaches R_7, which executes the same behavior as R_8 resulting in a packet sent with the Destination Address field set to the last one in the list, i.e. `FCBB:BBBB:0200::F001`. Finally, the packet reaches R_2 and triggers the End.DT6 behavior, which is responsible for removing the outer encapsulation header and forwarding the original packet to Site B.

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.

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. Considering the most common configuration, Micro SIDs are represented with 2 bytes, and up to 6 Micro SIDs can be carried in a regular 16 bytes SID, with a /32 Locator Block. For example, an SR path of 5 hops would require 16*5=80 bytes to be represented as a regular "SR path", while it would require 16 bytes to be represented as a sequence of 5 Micro SIDs encoded in a single regular SID. A detailed analysis of the overhead saving is discussed in Section VII. In this example, the non-compressed representation requires 40 bytes for IPv6 Header, 8 bytes for the fixed part of the SRH and 16*5=80 bytes for the SID list, in total 128 bytes. With the compressed representation we need to carry a single SID, which can fit in the IPv6 Destination Address with no need of using the SRH, only requiring 40 bytes for the IPv6 Header.

It is worth noting that using the Micro SIDs, whose size is much smaller than an IPv6 address, improves the performance by reducing the overhead (as shown later in Section VII). At the same time it is not affecting the overall scalability of the segment routing mechanism. Indeed, Micro SIDs are meant to work within a domain controlled by a network operator that allocates a /32 or /48 prefix to the Locator Block for Micro SID. In this domain, when uSIDs are represented with 16 bits, around 65k nodes can be deployed, which can support most typical transport backbones. As an example, a Mobile Network operator in Italy (Iliad) employs a total of about 10.5k nodes for its SRv6 deployment in a 5G national core network [14]. So, in this real-world deployment, the 16 bits for the uSID length would be sufficient to address all the nodes in the network. Anyway, to support larger transport networks, the Micro SID length can be increased reducing the saving in terms of compression efficiency (e.g. using 3 or 4 bytes), or the network can be partitioned in multiple Micro SID areas, similar to routing areas. In this work, we will consider that uSIDs are represented with 2 bytes.

As described in [6], the Micro SID solution proposes to extend SRv6 Network Programming with new behaviors, called uN, uA; they are the Micro SID variants of the plain SRv6 behaviors, as described in Table I. In particular, uN and uA are respectively used to replicate the End and End.X behaviors defined in not-compressed SRv6. As a quick tutorial on SRv6 architecture, we recall that the End behavior implements the logical function of "consuming" a segment and processing the next one, while the End.X behavior is a variant of the End behavior, which cross-connects the packet to a Layer-3 adjacency.

Plain SRv6	Micro SID
End	uN
End.X	uA

TABLE I: Plain SRv6 behaviors and Micro SID behaviors

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 uSID 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 uSIDs `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 "plain" shortest path IPv6 forwarding. As soon as R_8 receives the packet, it "consumes" its Micro SID identifier in the destination address: (i) the `0x0800` uSID is overwritten by the remaining uSID list, which is shifted left by 16 bits; (ii) 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 uSID `F00D` which triggers the final End.DT6 behavior: the packet is decapsulated and handled by a specific IPv6 routing table. In this example the uSID "`F00D`" encodes a function, in particular the End.DT6 behavior. Actually there are two types of uSIDs: *global* uSIDs that represent nodes to be reached and *local* uSIDs that represent a behavior to be executed. In [6] the two types are respectively called *Global* and *Local* uSIDs. *Global uSIDs* identify a shortest path to a specific node. *Local uSIDs* are instructions with local significance (e.g. End.DT6, End.DT4, End.DX6...). For each global SID (e.g. `0x0200`) a corresponding route (`FCBB:BBBB:0200::/48`) is advertised in the Micro SID domain. On the other hand the Local uSID do not need to be advertised to neighbor nodes from the reachability point of

view.

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 [15].

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. 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 [16].

To this end, we modified the table responsible for the SRv6 processing by adding the implementation of the Micro SID behaviors and the application logic to support the new uSID instructions. In particular, we developed the following extensions:

- added a new action named **srv6_usid_un**, responsible for (i) extracting the uSID of the next end router and (ii) updating the IPv6 destination address accordingly (see listing 1)
- added a new Longest Prefix Match (LPM) table named **srv6_localsid_table** responsible for both Micro SID and SRv6 processing (see listing 2)
- added a new action named **usid_ua** which applies the same shift-and-lookup logic as the uN behavior, but redirects the packet to a cross-connect table. In this way, the packet bypasses the global routing table by directly cross-connecting the packet to an output port identified in the uA instruction in the SID list.
- added the `End.X`, `End.DX6` and `End.DX4` actions that implement the various flavors of the End behaviors to support respectively the (i) Layer 2 cross-connect, (ii) decapsulation and IPv6 cross-connect and (iii) decapsulation and IPv4 cross-connect
- added the logic required to encapsulate an IPv4 or IPv6 packet with a uSID policy
- 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 [17].

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.

```
#define USID_BLOCK_MASK 0xffffffff << 96
action srv6_usid_un() {
    hdr.ipv6.dst_addr = (hdr.ipv6.dst_addr &
         USID_BLOCK_MASK) | ((hdr.ipv6.dst_addr << 16) &
          ~((bit<128>)USID_BLOCK_MASK));
}
```

Listing 1: uSID_un P4 code

```
table srv6_localsid_table {
    key = {
        hdr.ipv6.dst_addr: lpm;
    }
    actions = {
        srv6_end;
        srv6_end_x;
        srv6_end_dx6;
        srv6_end_dx4;
        srv6_usid_un;
        srv6_usid_ua;
        NoAction;
        }
    default_action = NoAction;
}
```

Listing 2: srv6_localsid_table P4 code

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 [18] 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 [19].

C. uSID VPP implementation

Virtual Packet processor (VPP) is an open source virtual router [20]. It implements a high-performance forwarder that can run on commodity CPUs. VPP often runs on top of

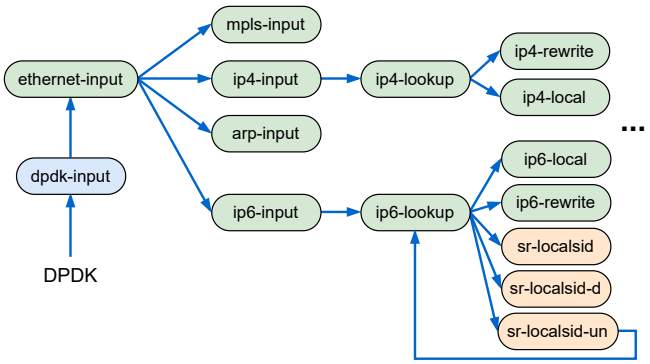

Fig. 2: VPP Packet Processing Architecture

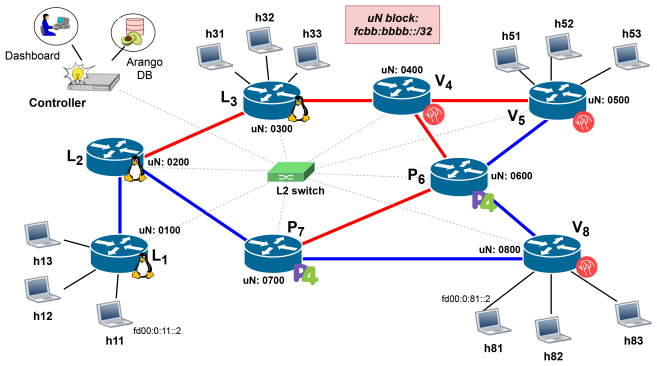

Fig. 3: uN Interoperability testbed network topology

the Data Plane Development Kit (DPDK) [21] 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. Fig. 2 shows an example of a VPP packet graph. VPP supports most of the behaviors defined in [1]. The behaviors are implemented in the *sr-localsid* and *sr-localsid-d* VPP graph nodes.

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 uN entry 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. As in any SRv6 functionality, the *sr-localsid-un* graph node also maintain counters to track traffic destined to such SID. Once the shifting is completed the packet is redirected to the (*ip6-lookup*) VPP graph node to determine the egress interface for the packet based on the updated IPv6 destination address.

V. 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 Micro SID primitives 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 [22]. Differently from this one, our demo is reproducible and publicly available at the project repository [17] and includes the detailed instructions to repeat the proposed experiments.

Figure 3 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 [20] (V_4, V_5, V_8);
- 2 programmable data planes nodes built on top of the software-based $P4_{16}$ implementation bmv2 [23] (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 Micro SID primitive set addressed by the proposed use case scenario consists of 3 functions. The first one is the `SR Source Node` 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 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.

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

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 VIII), 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 [24]. 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::/48`, where X is an index bound to the node in the range $[1, .., 8]$.

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::`.

Routing configuration. For sake of simplicity, the routing tables of the nodes are statically configured at startup by means of shell scripts.

The startup configuration script (which can not be listed in details for the limited space available for this publication) includes different kinds of routes:

1) routes toward the ipv6 addresses of all the routers links
2) routes toward the loopback interfaces of all the routers
3) routes toward the end hosts IPv6 addresses
4) routes to uN node performing the uN_un and the uN(End) functions
5) routes to perform the End.DT6 decapsulation function

Examples of the above mentioned routes are provided in listings 3, 4 and 5. It is worth noting that the 3 proposed implementations use different tools for configuring the internal routing tables: (i) the Linux kernel nodes exploit the `iproute2` suite; (ii) the VPP nodes are configured with the `vppctl` tool; (iii) the P4 nodes exploit the runtime controller client interface `simple_switch_CLI`.

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: $L_1(encap) \rightarrow L_2 \rightarrow P_7 \rightarrow P_6 \rightarrow V_5 \rightarrow V4_1(End) \rightarrow L_3(End.DT6)$. 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::`).

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

```
# Linux configuration

# link between routers
ip -6 route add fcf0:0:2:3::/64 via
      fe80::a41f:f2ff:fe53:d25a dev
      L1-L2 metric 20

# loopback interfaces
ip -6 route add fcff:2::/32 via
     fe80::a41f:f2ff:fe53:d25a dev
     L1-L2 metric 20

# end hosts
ip -6 route add fd00:0:11::/64 dev
     L1-h11 proto kernel metric 256

# uN behavior
ip -6 route add fcbb:bbbb:0100::/48
      encap seg6local action uN dev
      L2-L3

# End.DT6 behavior
ip -6 route add fcbb:bbbb:0100:f00d
     ::/80 encap seg6local action
     End.DT6 table 254 dev L1-h11
ip -6 route add fcbb:bbbb:0101::/64
      encap seg6local action End.
     DT6 table 254 dev L1-h11
```

Listing 3: Linux routes configuration

```
# VPP configuration
# link between routers
vppctl> ip route add fcf0
     :0:1:2::/64 via fe80::acec:4
     bff:fe20:4852 host-v5-p6

# loopback interfaces
vppctl> ip route add fcff:1::/32
     via fe80::e078:34ff:fe41:d6ec
     host-v5-v4

# end hosts
vppctl> ip6 table add 100
vppctl> ip route add fd00:0:51::/64
      table 100 via fd00:0:51::2
     host-v5-h51

# uN behavior
vppctl> sr localsid prefix fcbb:
     bbbb:0500::/48 behavior un 16

# End.DT6 behavior
vppctl> sr localsid prefix fcbb:
     bbbb:f00d::/64 behavior end.
     dt6 100
vppctl> sr localsid prefix fcbb:
     bbbb:0501::/64 behavior end.
     dt6 100
```

Listing 4: VPP routes configuration

```
# P4 configuration
# link between routers
simple_switch_CLI> table_add
      IngressPipeImpl.routing_v6
      IngressPipeImpl.set_next_hop
      fcf0:0:3:4::/64 => d2:8e:4f
      :23:63:59

# loopback interfaces
simple_switch_CLI> table_add
      IngressPipeImpl.routing_v6
      IngressPipeImpl.set_next_hop
      fcff:1::/32     => d2:8e:4f
      :23:63:59

# uN behavior
simple_switch_CLI> table_add
      IngressPipeImpl.
      srv6_localsid_table
      IngressPipeImpl.srv6_usid_un
      fcbb:bbbb:0700::/48 0
simple_switch_CLI> table_add
      IngressPipeImpl.
      srv6_localsid_table
      IngressPipeImpl.srv6_end
      fcbb:bbbb:0700::/64 0
```

Listing 5: P4 routes configuration

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.

VI. SDN/NFV USE CASE

A. High level description

This section focuses on a real world use case in which the SRv6/Micro SID programmability framework is used to allocate via an external SDN controller two network slices to support low latency and high throughput traffic. This use case is inspired by a recent work published by some of the authors of this paper [25].

We assume that a central controller is able to install a number of policies that bind a set of traffic classes to a set of segment routing paths. The traffic classification can be based on different strategies. For example, we can consider some arbitrary combinations of source and destination addresses, of transport layer protocol and ports as well as any other advanced traffic classification strategies. For sake of simplicity, as the actual traffic classification is outside the scope of this work, we are classifying the traffic by matching the source and destination addresses. The actual path computation to realize the required slices is realized offline for the specific topology. Other more complex and dynamic scenarios can be supported by leveraging standard policy based routing protocols. In any case, any change in the network topology or in the service requirements can be supported by recomputing the best paths and re-installing the SRv6 policies in the edge routers.

This use case is applied to the same topology deployed in the interoperability testbed. In this scenario, some links are characterized by a higher throughput, others by a lower latency, as evidenced with red (high throughput) and blue (low latency) links in Figure 3. In order to realize the slicing strategy mentioned above, the controller installs a set of SRv6 policy to the edge routers to enforce different paths according to the traffic classification. In this way, assigning a low latency slice or a high bandwidth slice is just a matter of pushing the proper SID list in the encapsulated packet at the edge.

B. Implementation

The first policy is meant to create a low latency network slice for the traffic between `h11` and `h51`. To achieve this goal, traffic has to go through only blue links which represent low latency links. The path for the low latency slice is L_1, L_2, P_7, V_8, P_6 and V_5. This path will be enforced by L_1 for traffic classified as low latency with an uSID policy. The policy will push a SID list that represents that path, but it is not needed to encode all the nodes of the path. In fact, the set of nodes needed in this case are L_2 (uN), P_7 (uN), V_8 (uN) and V_5 (uN and End.DX6) resulting into a single uSID container: `FCBB:BBBB:0200:0700:0800:0500:F00D::`. In this case all uSID IDs of the SRv6 policy fits in one uSID container which will be carried in the destination address of the outer IPv6 header of the SRv6 packet.

The second policy is meant to create a high bandwidth network slice for the traffic between `h12` and `h52`. Thus, traffic has to go through only red links which represent high bandwidth links. The candidate path for the high bandwidth slice is L_1, L_2, L_3, V_4 and V_5 . This path will be enforced by L_1 for traffic classified as high bandwidth with an uSID policy. Again, the set of nodes needed are less than the actual path: L_3 (uN) and V_5 (uN and End.DX6). The resulting policy will fit in just one uSID container: `FCBB:BBBB:0300:0500:F00D::`. Also here the uSID list will be carried in the destination address of the outer IPv6 header of the SRv6 packet.

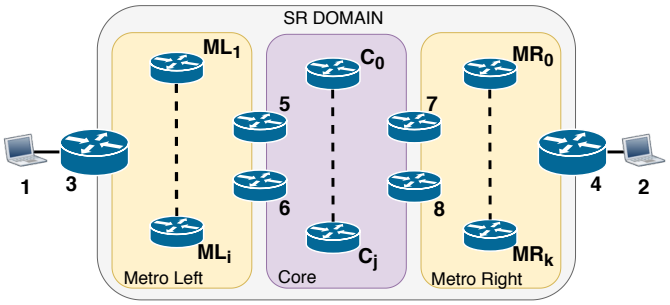

Fig. 4: Reference topology for Compression Analysis

The project software repository contains the configuration scripts required to deploy the proposed use case.

VII. Evaluation of compression savings

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 list 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) [8].

We show that a mapping solution (like SRm6) does not provide better compression than what can be achieved with the Micro SID mechanism. As such, analysis of the SRm6 proposal documented in [8] is provided for comparison.

The SRm6 solution [8] defines a new routing header called *Compact Routing header* (CRH) to be used to carry the list of segments instead of the SRH. More specifically, [8] 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.

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. 4. 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 ($\mathrm{ML_1..ML_i}$), Core ($\mathrm{C_1..C_j}$), and Metro R ($\mathrm{MR_1..MR_k}$). 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 [5], 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 header (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 an 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 IPv6 in IPv6 packet without the SRH extension header.

According to [26], 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 bits represent the node identifier along with the function to be applied. The Argument bits contain the uSIDs in the micro-program.

A 16-byte SRv6 instruction that contains a micro-program is called a uSID *container* instruction and has the structure shown in Figure 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) [9]. The CRH is made of a fixed set

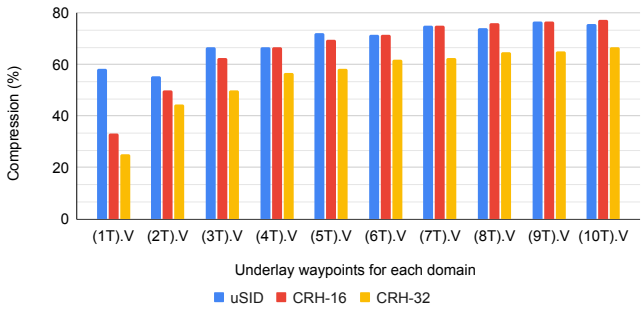

Fig. 5: Encapsulation size Saving for uSID and SRm6

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

$$E_{CRH16}(SL) = 40$$

if $|SL| = 1$, otherwise

$$E_{CRH16}(SL) = 40 + \lceil (4 + |SL| * 2)/8 \rceil * 8.$$

In CRH-32 the length of headers is calculated as

$$E_{CRH32}(SL) = 40$$

if $|SL| = 1$, otherwise

$$E_{CRH32}(SL) = 40 + \lceil (4 + |SL| * 4)/8 \rceil * 8.$$

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, considering the case without the Destination Options header called TPF (Tunnel Payload Forwarding) proposed in [27]. Note that in a multi domain scenario the CRH solution requires one additional SID at each domain boundary (two in our reference scenario). For simplicity these additional SIDs have been neglected in our evaluation, resulting in a small overestimation of the savings that can be achieved with CRH.

Figure 5 plots the Encapsulation size Saving for the three solutions, considering the reference scenario in a range from one to 10 underlay waypoints for each domain (Metro L, Core, Metro R), resulting in a list of 31 SIDs. The Micro SID compression is significant (58%) also for a SID list of just 4 nodes (first group of bars in Figure 5), 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 75%. It is worth underlining that in the case of 8 and 10 waypoints, the Micro SID mechanism performs slightly worse (nearly 1-2%) with respect to CRH-16. This depends on how the uSIDs are encoded in IPv6 addresses. Considering a 16 bit uSID and a 32 bit Locator Block, an IPv6 address permits to encode 6 uSIDs. When the number of uSIDs in the list is a multiple of 6, the terminator uSID (e.g. "`F00D`") requires an entire IPv6 address. Consequently, in this case, the remaining unused bits in the IPv6 address increase the overhead of our solution.

VIII. Performance Assessment

This section presents a performance analysis of the Micro SID header compression mechanisms based on a set of standalone 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 implementation, we have reserved two bare metal servers on the federated testbed infrastructure CloudLabs [28]. 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 [29] runs on the TG machine.

The two reserved servers are x86 machines equipped with 2x Intel E5-2630 v3 85W 8C at 2.40 GHz for a total of 16 cores in hyper-threading, 32KB L1 cache, 256KB L2 cache, 20MB L3 cache, 128 GB of ECC RAM (8x 16 GB DDR4 2133 MHz PC4-17000 dual rank RDIMMs) and the Intel Ethernet Network Adapter X520 - Dual Port 10Gb SFP+ (connected via PCIe v3.0, 8 lanes). On the SUT machine we have installed a Linux 5.6 kernel patched with our Micro SID implementation.

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 an SRH containing two SIDs (158 bytes);
4) function `uN(End)` on IPv6 packets plus an SRH containing two SIDs (158 bytes);
5) function `uA(ua)` on IPv6 packets plus an SRH containing two SIDs (158 bytes);
6) function `uA(End)` with IPv6 packets plus SRH containing two SIDs (158 bytes);
7) function `End` (basic SRv6) on IPv6 packets with an SRH containing two SIDs. Such behavior is considered to be our performance baseline for the uN behavior. The other experiments are compared to this one to understand the

#	Function	Encap	PDR@0.5%	Perf. Gain
1	uN(un)	IPv6 in IPv6	869.61 kpps	+2.48%
2	uN(un)	IPv6 in IPv6 (pad)	869.66 kpps	+2.48%
3	uN(un)	IPv6 + SRv6	861.52 kpps	+1.52%
4	uN(End)	IPv6 + SRv6	843.17 kpps	-0.64%
5	uA(ua)	IPv6 + SRv6	127.62 kpps	-0.37%
6	uA(End)	IPv6 + SRv6	128.62 kpps	+0.40%
7	End	IPv6 + SRv6	848.60 kpps	———
8	End.X	IPv6 + SRv6	128.10 kpps	———

TABLE III: Linux kernel performance assessment

overhead introduced by the proposed header compression mechanism. The packet size for this experiment is 158 bytes;

8) function `End.X` (basic SRv6) on IPv6 packets with an SRH containing two SIDs. As the End behavior, such behavior is considered to be our performance baseline for the uA behavior. Also in this experiment, the packet size is 158 bytes.

B. Linux kernel implementation assessment

The detailed results of the above described experiments for the Linux kernel Micro SID 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 reported in Table III is measured in Packets Per Second and it is meant to show the processing performance overhead introduced by Micro SID with respect to the standard SRv6 processing. The throughput (848.60 kpps) measured in the experiment 7 is taken as reference to evaluate the increase or decrease in performance experienced in the experiments concerning the uN behavior; the throughput measured in the experiment 8 is taken as the baseline for the experiments concerning the uA behavior. Indeed, the SRv6 End and End.X behaviors do not perform any uSID 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 [30] and [31]. PDR is the highest throughput achieved without dropping more than a predefined threshold, in this case 0.5% of the overall packets. The PDR allows to characterize a given configuration of a System Under test with a single scalar value, without considering the relation between incoming rate and achieved throughput.

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,

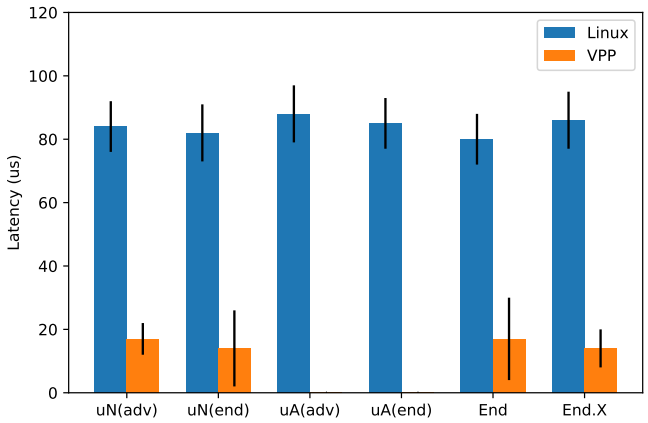

Fig. 6: Latency assessment of SRv6/MicroSID behaviors on Linux and VPP implementation

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.

The performance of the uA behavior with respect to the baseline (SRv6 End.X) shows similar results as uN: in the case on `uA(ua)` we obtain a tight performance drop of 0.37% while in the case of `uA(end)` the performance improves of 0.40%. The experienced throughput for uA experiments is around 128kpps that is significantly worse with respect to uN experiments. This is motivated by the fact that the implementation of SRv6 End.X behavior in the Linux kernel suffers a significant performance drop with respect to other SRv6 behaviors. Since the uA behavior is based on End.X, the obtained performance is in the same order of magnitude.

These results highlighted 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[21], 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. For what concerns the uA behavior, the implementation for VPP is planned, but since no user has actively requested this feature we have not started its implementation yet.

D. Linux and VPP latency assessment

In this subsection, we discuss the latency assessment for the Linux and VPP implementation. The testbed and testing

scenario is the same as discussed in Section VIII-A.

Latency measurement tool. To assess the latency, in the TG we did not use the TRex traffic generator but a custom C program that provides more accurate timestamps, since it employs the TX/RX hardware timestamping functionalities of the NIC. The source code of the program is available online at [32]. The latency test consists in sending a stream of packets from the TG, rated at one packet per second (so under low traffic conditions), that is received and processed with the tested Micro SID behavior by the SUT. Then the probe packet is sent back towards the same interface it was received from. The latency measurement is calculated by the TG by evaluating the difference between the hardware timestamp registered in TX, with the one registered in RX. In these experiments, the probe packet used for the latency tests always carries PTPv1 (Precision Time Protocol) data, that was necessary in order to apply the hardware timestamp in RX. In fact, the NICs employed in the testbed do not support RX hardware timestamping on arbitrary packets but only on PTP messages.

Figure 6 represents the average of 60 experiments of the measured latency and the standard deviation for each of the considered Micro SID behaviors in the Linux and VPP implementations. These tests highlighted that, in concordance with the throughput measurements, the performance in terms of end-to-end latency is significantly better in VPP, since it employs the DPDK framework with kernel bypass. In fact, the average score for VPP is about $14\mu s$ while in the Linux implementation we obtained around $80\mu s$.

E. P4 implementation assessment

As described in Section IV, 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 [16], 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 IV is based on $P4_{16}$ and may not be compatible with existing hardware like Tofino [33] as is. In particular the uN and uA behaviors cannot be written in $P4_{14}$ as described in the Listings reported in the extended version, but must be segmented through multiple pipeline stages.

IX. Related works

A. SRv6 protocol extensions and optimizations

A comprehensive survey of the research on SRv6 can be found in [34]. Among all the reported literature works, a considerable number is related to our work, like the ones focusing on optimizations [35][36][37][38]. A survey of the SRv6 use cases can be found in [39].

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 [10][8][9][11].

The authors in [7] describe a compressed version of the SID list (C-SID) and SRH (C-SRH) that would decrease significantly the overhead carried by SRv6. Anyway, this mechanism requires some modifications in data plane, while our Micro SID solution builds upon the base SRv6 processing and needs minimal additions to existing solutions.

The COC solution is defined in the context of the framework called "Generalized SRv6 Network Programming for SRv6 Compression" (G-SRv6) [10]. 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 [15], wherein uSID and COC are formally defined as extensions of SRv6 End and End.X flavors. The two new flavors are respectively called NEXT-C-SID (uSID) and REPLACE-C-SID (COC) in [15] and they can be used stand-alone or in their combination, referred to as NEXT-REPLACE-C-SID.

[8] and [9] propose 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). As shown in Section VII, this approach provides similar compression benefits to our solution, but SRm6 needs a new control and data plane, a new ecosystem (not SRv6-native) and additional lookups at egress PE [26].

The work described in [11] 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.

Overall, all the considered solutions provide similar benefits in terms of compression efficiency. Anyway, Micro SID is the only solution that enables SID compression maintaining full compatibility and seamless interoperability with existing SRv6 data plane or control plane. As a result, the data plane and control plane performance is intrinsically the same as SRv6, as demonstrated in Section VIII.

C. Segment Routing in SDN/NFV scenarios

SRv6 has been proved to be particularly suited for SD-N/NFV scenarios [40]. Abdelsalam et al. [41] 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 [42]. The performance of the Linux's SRv6 implementation has been assessed in [43].

Other relevant SW based implementations [44], [45] 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 [45] to realize in-network programmability use cases. Moreover, an implementation of SRv6 on P4 dataplane and ONOS controller has been presented in a tutorial[16].

SR has been also exploited in SDN scenarios. Ventre et al. [46] presented an SDN Architecture and Southbound APIs for IPv6 Segment Routing Enabled Wide Area Network application scenarios. Bidkar et al. [47] presented an SDN framework built upon Carrier Ethernet and augmented with SR. L. Huang et al. [48] provide a novel SR architecture based on OpenFlow that reduces the overhead of additional flow entries and label space. Dugeon et al. [49] implement and assess the SR approach with SDN based label stack optimization on top of the SDN controller OpenDaylight. Lee et al. [50] propose a routing algorithm for SDN with SR that can meet the bandwidth requirements of routing requests. Among all works cited in this survey, a considerable number focuses on SRv6 deployment in NFV/SDN scenarios.

X. 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 introduced the basic approach, the encoding mechanism and the detailed operations for different SRv6 behaviours. 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, P. Camarillo, J. Leddy, D. Voyer, S. Matsushima, and Z. Li, "Segment Routing over IPv6 (SRv6) Network Programming," RFC 8986, Feb. 2021. [Online]. Available: https://rfc-editor.org/rfc/rfc8986.txt

[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, C. Xie, R. Bonica, D. Dukes, C. Li, S. Peng, and W. Henderickx, "Compressed SRv6 SID List Requirements," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-compression-requirement-01, Mar. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-ietf-spring-compression-requirement/01/

[6] C. Filsfils, P. Camarillo, D. Cai, D. Voyer, I. Meilik, K. Patel, W. Henderickx, P. Jonnalagadda, D. T. Melman, Y. Liu, and J. Guichard, "Network Programming extension: SRv6 uSID instruction," Internet Engineering Task Force, Internet-Draft draft-filsfils-spring-net-pgm-extension-srv6-usid-12, Dec. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-filsfils-spring-net-pgm-extension-srv6-usid-12

[7] Z. Li, C. Li, C. Xie, K. LEE, H. Tian, F. Zhao, J. Guichard, L. Cong, and S. Peng, "Compressed SRv6 Network Programming," Internet Engineering Task Force, Internet-Draft, Aug. 2020, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-li-spring-compressed-srv6-np-02

[8] R. Bonica, S. Hegde, Y. Kamite, A. Alston, D. Henriques, L. Jalil, J. M. Halpern, J. Linkova, and G. Chen, "Segment Routing Mapped To IPv6 (SRm6)," Internet Engineering Task Force, Internet-Draft, Sep. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-bonica-spring-sr-mapped-six-04

[9] R. Bonica, Y. Kamite, A. Alston, D. Henriques, and L. Jalil, "The IPv6 Compact Routing Header (CRH)," Internet Engineering Task Force, Internet-Draft, Nov. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-bonica-6man-comp-rtg-hdr-27

[10] W. Cheng, Z. Li, C. Li, F. Clad, A. Liu, C. Xie, Y. Liu, and S. Zadok, "Generalized SRv6 Network Programming for SRv6 Compression," Internet Engineering Task Force, Internet-Draft, Oct. 2021, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-cl-spring-generalized-srv6-for-cmpr-04

[11] 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-07, Mar. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-decraene-spring-srv6-vlsid-07

[12] A. Tulumello, A. Mayer, M. Bonola, P. Lungaroni, C. Scarpina, S. Salsano, A. Abdelsalam, P. Camarillo, D. Dukes, F. Clad, and C. Filsfils, "Micro sids: a solution for efficient representation of segment ids in srv6 networks," in *2020 16th International Conference on Network and Service Management (CNSM)*, 2020, pp. 1–10.

[13] 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.

[14] S. Matsushima, C. Filsfils, Z. Ali, Z. Li, K. Rajaraman, and A. Dhamija, "SRv6 Implementation and Deployment Status," Internet Engineering Task Force, Internet-Draft draft-matsushima-spring-srv6-deployment-status-15, Apr. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-matsushima-spring-srv6-deployment-status-15

[15] W. Cheng, C. Filsfils, Z. Li, B. Decraene, D. Cai, D. Voyer, F. Clad, S. Zadok, J. Guichard, A. Liu, R. Raszuk, and C. Li, "Compressed SRv6 Segment List Encoding in SRH," Internet Engineering Task Force, Internet-Draft draft-ietf-spring-srv6-srh-compression-01, Mar. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-srv6-srh-compression-01

[16] 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

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

[18] "iproute2 website," https://wiki.linuxfoundation.org/networking/iproute2.

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

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

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

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

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

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

[25] 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.

[26] R. Bonica, W. Cheng, D. Dukes, W. Henderickx, C. Li, S. Peng, and C. Xie, "Compressed SRv6 SID List Analysis,"

Internet Engineering Task Force, Internet-Draft draft-ietf-spring-compression-analysis-01, Mar. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/html/draft-ietf-spring-compression-analysis-01

[27] R. Bonica, Y. Kamite, L. Jalil, Y. Zhou, and G. Chen, "The IPv6 Tunnel Payload Forwarding (TPF) Option," Internet Engineering Task Force, Internet-Draft draft-bonica-6man-vpn-dest-opt-18, Jul. 2022, work in Progress. [Online]. Available: https://datatracker.ietf.org/doc/draft-bonica-6man-vpn-dest-opt/18/

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

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

[30] 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.

[31] 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.

[32] A. Tulumello. Hardware timestamp utility repository. [Online]. Available: https://gitlab.com/angelo.tulumello/hw-ts-util

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

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

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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

[40] 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

[41] 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.

[42] 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

[43] 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.

[44] S. Goldshtein, "The next linux superpower: Ebpf primer," *Dublin: USENIX Association*, 2016.

[45] 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

[46] P. L. Ventre, M. M. Tajiki, S. Salsano, and C. Filsfils, "Sdn architecture and southbound apis for ipv6 segment routing enabled wide area networks," *IEEE Transactions on Network and Service Management*, vol. 15, no. 4, pp. 1378–1392, 2018.

[47] 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.

[48] 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.

[49] 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.

[50] 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.