TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA

cnit consorzio nazionale interuniversitario per le telecomunicazioni

SRV6 WORKSHOP @ NETDEV 0X1A

SRv6 Layer 2 Services support the sr6 device and End.DT2U

Andrea Mayer[1], Stefano Salsano[1,2]

[1]University of Rome Tor Vergata, Italy · [2]CNIT, Italy

Netdev 0x1A — The Technical Conference on Linux Networking

SRv6 Workshop — *13 July 2026, Rome, Italy*

Work developed together with Ahmed Abdelsalam and Clarence Filsfils (Cisco Systems)

Funded by the European Union
NextGenerationEU

PRIN
Ministero dell'Università e della Ricerca

Italia Domani
National Recovery and Resilience Plan

PRIN 2022
NEWTON

The story in three messages

1

The gap is in Linux, not in SRv6.

RFC 8986 already defines the L2 behaviors — but Linux has no bridge-facing L2 device like `vxlan`, so ARP and non-IP never cross the overlay.

2

The fix: the `sr6` device + `End.DT2U`.

A bridge-integrated Ethernet pseudowire that makes SRv6 L2 overlays *operationally equivalent to VXLAN* in Linux (for point-to-point).

3

No protocol change — and a smaller wire than VXLAN.

The L2 service ID rides as a standard SRv6 µSID in the Destination Address: **40 B** vs 56 B of VXLAN. Landing in RFC v2 before submission.

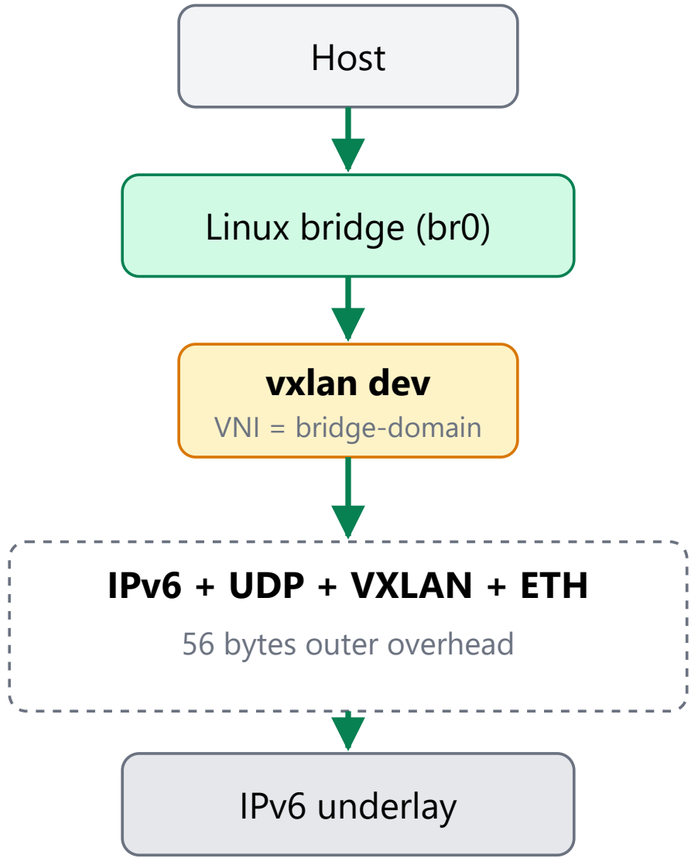
THE OPERATIONAL REFERENCE

VXLAN: the Linux operational reference

Why VXLAN works in Linux:

- **Native netdevice** attached to a bridge
- **L2-triggered** encap, not IP-routed
- All Ethernet crosses — **ARP included**
- **24-bit VNI** = the bridge domain

The bar to match. Direct bridge integration + an explicit L2 service identifier — the model behind EVPN, Neutron, CloudStack.



PART 1 — THE IMPLEMENTATION GAP

SRv6 already covers Layer 2 — architecturally

Behavior	What it does (RFC 8986)	In Linux today
`End.DX2`	Decap & forward the inner Ethernet frame to a **specific outgoing L2 interface**	✓ seg6local — cross-connect only
`End.DT2U`	Decap & deliver into a **local Ethernet domain**, unicast MAC lookup	✗ → **this patch series**
`End.DT2M`	Decap & **flood** into the L2 domain (BUM traffic)	✗ → next step

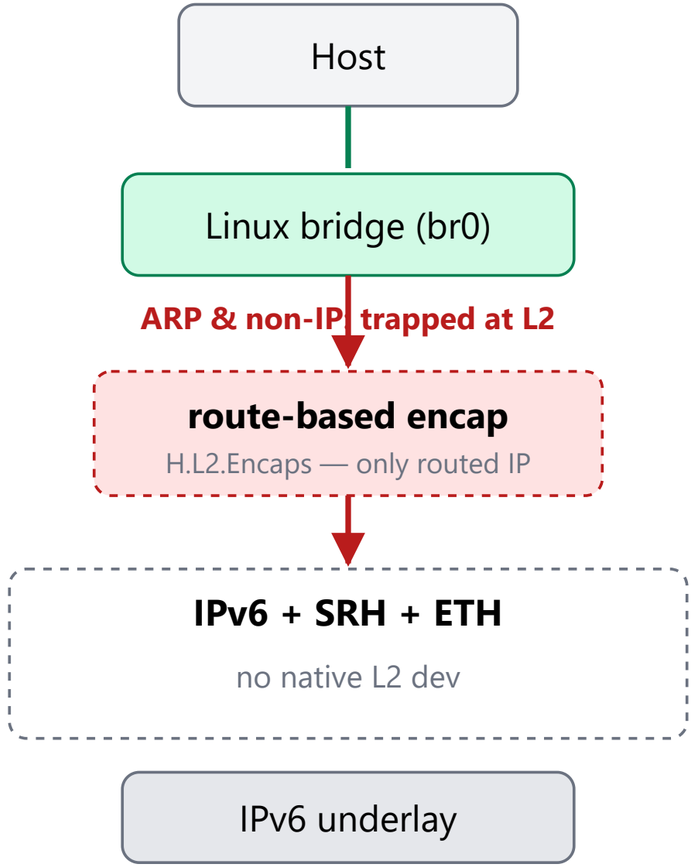
A behavior specifies packet processing — not host integration. RFC 9252 puts EVPN on top of these behaviors; none of them says how the endpoint becomes a *netdevice* and joins the local Ethernet data plane.

Current Linux SRv6 L2: the implementation gap

The behaviors exist — Linux does not expose them for bridging:

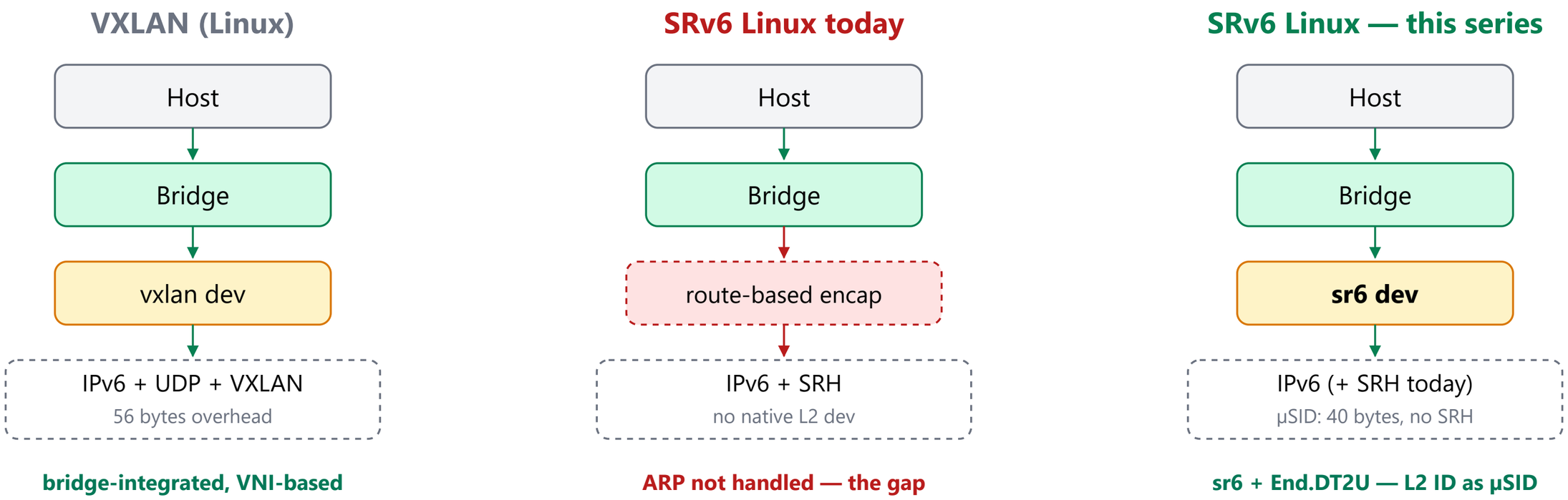
- ✗ No native **L2 netdevice** to attach to a bridge
- ✗ Encap is **route-based** (`H.L2.Encaps`, IP only)
- ✗ **ARP & non-IP** never reach the encap point
- ✗ No bridge-integrated, transparent L2 service

A Linux gap — not an SRv6 gap. The architecture is complete; the bridge-facing wiring is missing.



PART 2 — CLOSING THE GAP

Design choice: close the gap, don't extend SRv6



the reference — the gap — the bridge-integrated SRv6 model

Two decisions. (1) Add a native bridge-integrated `sr6` device + Linux `End.DT2U`; (2) carry the L2 service ID inside the SRv6 programming model — *no new architectural element*.

PART 3 — THE PATCH SERIES

The RFC v2 series in one slide

Kernel — 4 patches

- `seg6`: SRv6 L2 tunnel device (`sr6`)
- `seg6`: SRv6 `End.DT2U` behavior
- `selftests`: sr6 + End.DT2U L2 VPN test
- `docs`: `Documentation/networking/sr6.rst`

`CONFIG_IPV6_SR6` (module `sr6`) · depends on `IPV6_SEG6_LWTUNNEL`, selects `DST_CACHE` · netlink spec in `rt-link.yaml`

iproute2 — 5 patches

- `sr6` link type · `End.DT2U` action
- optional SRH **HMAC** (`hmac KEYID`)
- man pages: `ip-link`, `ip-route`

UAPI: `IFLA_SR6_SRH`, `IFLA_SR6_FIB_TABLE` · `SEG6_LOCAL_ACTION_END_DT2U` (17), `SEG6_LOCAL_L2DEV`

Status. µSID / reduced encapsulation is the last item in progress — the series goes to `netdev@` as soon as it lands.

PART 3 — THE PATCH SERIES

The `sr6` device: an Ethernet pseudowire netdevice

- rtnl link type `sr6` — **attaches to a bridge**
- Takes L2 frames → outer IPv6, `IPPROTO_ETHERNET`
- Segment list & FIB table **immutable** after creation
- Per-cpu `dst_cache`, invalidated on l3mdev change
- No netns move · VLAN only with an explicit `table`

Same role for SRv6 that the `vxlan` netdevice plays for VXLAN.

On the wire (today: SRH-based encap)

outer encap — the inner L2 frame is carried unchanged

overhead = 40 + (8 + 16·n) = 48 + 16·n

µSID / reduced encap: no SRH → 40 B
in progress — lands in RFC v2 before submission

PART 3 — THE PATCH SERIES

`End.DT2U` in `seg6local`

- New action **17** + attribute `SEG6_LOCAL_L2DEV`
- Decap `IPPROTO_ETHERNET` → `netif_rx()` on `l2dev`
- `l2dev` = **bridge port** or another `sr6` device
- Enforced at `build_state`, with a proper `extack`
- The bridge does MAC learning — that is the **U** in DT2U

The frame re-enters a valid L2 context. Restricting `l2dev` is what keeps true Ethernet-overlay semantics.

```
/* RFC 8986 requires L2 forwarding semantics.
 * Only bridge ports and sr6 devices satisfy it. */
static bool seg6_is_valid_l2dev(const struct net_device *dev)
{
    return netif_is_bridge_port(dev) || netif_is_sr6(dev);
}
```

The subtle one

`skb_scrub_packet()` must run **before** `eth_type_trans()`. Scrubbing afterwards resets `pkt_type` to `PACKET_HOST` — and multicast / broadcast silently break on the bridge.

End-to-end operation

today: SRH carries the segment list · roadmap: µSIDs in the DA, no SRH

outer DA = locator + behavior + L2 service ID

Encap is triggered by **L2 forwarding**, not IP routing

⇒ **all** Ethernet crosses the overlay: IPv4, IPv6, **ARP**, non-IP

On egress, **`l2dev`** = bridge port or another sr6 device

Operationally equivalent to VXLAN at the Linux level. The host-facing interface and the `sr6` device are members of the same bridge; ordinary L2 forwarding does the rest.

ARP & non-IP: why it is a *true* Ethernet overlay

Route-based encap only sees packets on the IP routing path:

- ✗ ARP is handled at L2 — **never encapsulated**
- ✗ non-IP protocols are dropped from the overlay

Bridge-integrated `sr6` — ARP follows the path of any frame:

- ✓ flooded / forwarded by ordinary L2 logic
- ✓ reaches `sr6` → encapsulated into the underlay
- ✓ remote `End.DT2U` delivers it into the L2 domain

> **Why it matters**
>
> A transparent Ethernet service requires *broadcast / unknown-unicast / multicast* and address resolution to cross the overlay — exactly what a routed-IP tunnel cannot do.

The datapath is driven by L2 forwarding, not by IP-route interception. That is what turns a tunnel into a genuine Ethernet overlay.

PART 4 — KERNEL DETAILS THAT MATTER

Post-encap routing: four lookup modes

Mode	VRF context	`sr6` table	Where the SID route must live
plain	no	—	`main` — resolved through fib rules
VRF	yes	—	the **VRF table** — blackhole in main prevents leaking
table	no	X	table **X** — direct lookup, **fib rules bypassed**
VRF + table	yes	X	table **X** — the explicit table **wins** over the VRF

All four are exercised by the selftest. The `dst_cache` is per-cpu and is invalidated automatically when the l3mdev context changes — a bridge can be moved in and out of a VRF at runtime.

PART 4 — KERNEL DETAILS THAT MATTER

Configuration in practice

```
# ingress PE — create the pseudowire and put it on the bridge
ip link add sr6-0 type sr6 segs fcff:2::d20
ip link set sr6-0 up
ip link set sr6-0 master br0        # same bridge as the host-facing port

# multi-hop path with an explicit FIB table
ip link add sr6-1 type sr6 segs fcff:3::e,fcff:2::d20 table 200

# egress PE — the local SID that terminates the L2 VPN
ip -6 route add fcff:1::d20 table 90 \
    encap seg6local action End.DT2U l2dev sr6-0 dev dum0
```

No new userspace model. `sr6` is an ordinary link type, `End.DT2U` is an ordinary `seg6local` action — both documented in the `ip-link` and `ip-route` man pages of the series.

Beyond bridging: standalone, VRF, IRB

Standalone.

- No bridge: IP addresses go directly on `sr6` — a point-to-point L3 interface over the L2 tunnel. `End.DT2U` already accepts an `sr6` as `l2dev`, so the remote SID does not change.

VRF.

- Enslave the bridge to a VRF: `sr6` inherits the routing context and performs the route lookup for the encapsulated packet in the VRF table — per-service routing domains, for free.

IRB (Integrated Routing and Bridging).

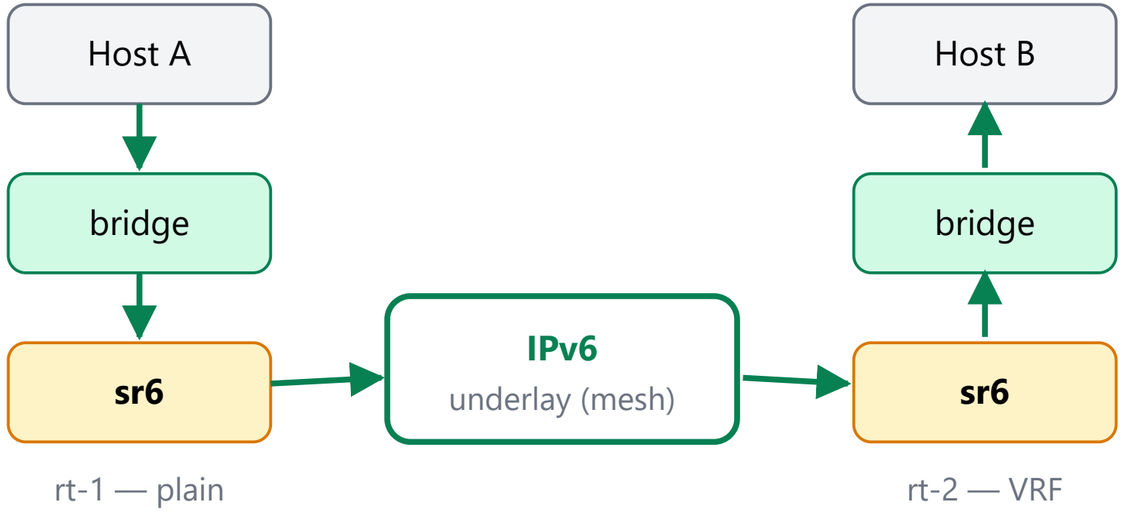
★ Give the bridge an address inside the VRF: the PE becomes the L3 gateway of the overlay and routes to other subnets *without leaving the box*.

All three documented in `Documentation/networking/sr6.rst`, with full configurations.

PROTOTYPE VALIDATION

`srv6_sr6_l2vpn_test.sh`: the full datapath, exercised

- **4 SRv6 routers**, full-mesh IPv6 underlay
- One host each: bridge + `sr6` as a bridge port
- **2 independent L2 VPNs** — cross-VPN isolation checked
- Segment lists of **1, 2 and 3 SIDs** (transit `End`)
- A different **lookup mode per router** — all four
- Blackhole routes prove the right table is used
- Checks host↔host, host↔remote gw, router↔router



2 L2 VPNs over 4 routers

rt-3: table · rt-4: VRF + table

1 / 2 / 3-SID paths, transit End

cross-VPN traffic must NOT pass

ARP crossing the VPN is the proof — the datapath is driven by bridge L2 forwarding, not by IP-route interception.

Wire format: smaller than VXLAN, plain IPv6 on the wire

Stack (IPv6 underlay)	Header chain	Outer overhead
VXLAN over IPv6	IPv6 + UDP + VXLAN + ETH	56 bytes
SRv6 L2 — RFC v2 branch today	IPv6 + SRH + ETH	64 bytes (1 SID)
SRv6 L2 — µSID / reduced (landing in RFC v2)	IPv6 + ETH — **no SRH**	**40 bytes**

The L2 service ID is just one more µSID in the same Destination Address. 16 bytes saved per packet against VXLAN, transit nodes do an ordinary IPv6 DA lookup — and the overlay crosses *any* IPv6 underlay. No SRv6 protocol change, no architectural extension.

PART 5 — WHAT COMES NEXT

What is missing — and what we ask of this room

1 **`End.DT2M`: multipoint services.**
BUM replication and split-horizon are the steps from point-to-point pseudowires to a real E-LAN service.

2 **Control plane.**
EVPN over SRv6 (RFC 9252) in FRR — SID allocation and MAC/IP advertisement on top of the datapath we are landing now.

3 **Consumers.**
Replacing VXLAN where it actually runs: Kubernetes CNIs, OVN, OpenStack Neutron — they all want a bridge-facing device, and now there is one.

4 **Performance.**
Systematic evaluation against VXLAN — and your review of the datapath before the series hits the list.

Conclusions

Contribution

`sr6` + `End.DT2U` close the Linux gap: bridge-integrated, ARP-capable, **VXLAN-equivalent** SRv6 L2 overlays. 4 kernel + 5 iproute2 patches, with selftest and documentation.

Next

µSID / reduced encapsulation (40 B, no SRH), then **RFC v2 to `netdev@`**. After that: `End.DT2M`, EVPN over SRv6, cloud orchestrators.

Take-home. SRv6 Layer-2 services are realizable in Linux *today* — the architecture was never the problem, the netdevice was.

Review welcome, before it hits the list:

`github.com/netgroup/srv6-linux-kernel-dev` → `seg6_sr6_L2encap_rfc-v2`

`github.com/netgroup/srv6-linux-iproute2-dev` → `seg6_sr6_L2encap_UAPI-v2`

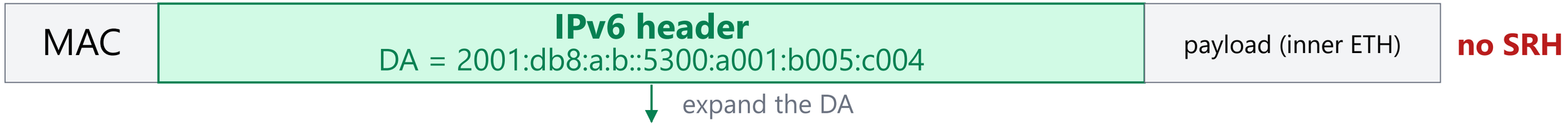
`andrea.mayer@uniroma2.it` · `stefano.salsano@uniroma2.it`

BACKUP

SRv6 micro-SIDs in one slide

One 128-bit IPv6 destination address carries a locator **plus a sequence of compact instructions** — no SRH needed.

Encapsulated packet



Inside the destination address (128 bits)

Each SRv6 hop **consumes the leftmost µSID** and shifts the next instruction left.

2001:db8:a:b::5300:a001:b005:c004 → 2001:db8:a:b::a001:b005:c004:0 → ...

One DA holds many instructions — no SRH, smaller overhead, plain IPv6 lookup at every transit hop.