Challenges and limitations of debugging virtualized networks with Retis

Antoine Ténart Paolo Valerio Adrián Moreno

Red Hat

Introducing retis

We have written a network troubleshooting tool called **retis**.

Challenges / requirements

- Lots of data and data sources
- Lots of packets
- Packets mutate
- Packets sometimes go through userspace: OVS

Example #0

```
$ retis collect -c skb -p tp:net:netif_receive_skb \
                       -p tp:net:net_dev_start_xmit

431586745164219 (11) [ping] 3797350 [tp] net:net_dev_start_xmit

  if 483 (pint) 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0 id 12810 off 0 len 84
proto ICMP (1) type 8 code 0


431586745198007 (11) [ping] 3797350 [tp] net:netif_receive_skb

  if 482 (local_pint) 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0 id 12810 off 0 len
84 proto ICMP (1) type 8 code 0
```

Lots of data and data sources

… in the net stack; what does that mean?

Collecting packets (using eBPF) from various points: using different probe types – k(ret)probes, (raw) tracepoints, fentry/fexit, etc.

- Data access is not uniform (e.g. probe arguments)
- Data is in an unknown state: no assumption (e.g. ***struct sk_buff*** fields uninitialized, pointers not updated yet)
- Target data (e.g. ***struct sk_buff***) not always directly accessible
- Not all helpers available (e.g. ***bpf_get_func_ip***)
- Different limitations (e.g. no freplace in fentry/fexit, no entry args in kretprobe)

Need for a uniformization layer

And select probe types based on requirements – or have workarounds.

- Dynamically parse probe arg types (at load time)
- Providing custom getters to handle indirections (e.g. ***nft_pktinfo->skb***)
- Using a kprobe/kretprobe pair to access entry args
- Providing the func_ip equivalent in tracepoints (we use it as a config map key)

Uniformization goes beyond the above:

- To support different kernel versions (in most cases CO-RE helps)
- To support different architectures (e.g. ***PT_REGS_IP(ctx) - 1*** on x86_64)
- System state is unknown (e.g. modules loaded)
- Compiler might inline functions
- Etc

But also various components are traversed

Information retrieval is split among different collectors:

- ct, nft, ovs, skb, skb_drop, skb_tracking
  - Some collectors, when enabled, implicitly request to attach to specific kernel functions or tracepoints
    - nft => ***__nft_trace_packet()***
    - skb_drop => ***skb:kfree_skb()***
    - skb_tracking, ovs => ...
  - The other collectors simply extend all the programs (i.e. ct, skb)
  - All send data to user-space through a shared ringbuffer

```
struct {
        __uint(type, BPF_MAP_TYPE_RINGBUF);
        __uint(max_entries, sizeof(struct retis_raw_event) * EVENTS_MAX);
} events_map SEC(".maps");
```

But also various components are traversed

Collectors extend the main program:

- The eBPF programs include a series of calls to stub functions
- Stubs are then replaced at load time

```
#define HOOK(x)                                                                 \
        __attribute__ ((noinline))                                              \
        int hook##x(struct retis_context *ctx, struct retis_raw_event *event) { \
                volatile int ret = 0;                                           \
                if (!ctx || !event)                                             \
                        return 0;                                               \
                return ret;                                                     \
        }

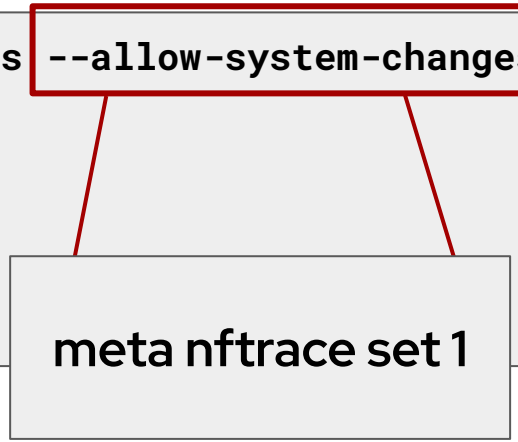
#define CALL_HOOK(x)                            \
        if (x < nhooks) {                       \
                int ret = hook##x(ctx, event);  \
                if (ret == -ENOMSG)             \
                        goto discard_event;     \
        }
```

What's next?

- Initial motivation for freplace: custom hooks
  - Theoretically simplifies hooks precedence
- No more custom hooks?
- Toward inlining (RFC)
  - This plays well with fentry/fexit

Example #1

```
$ retis collect -c skb,nft,ct --skb-sections dev,ns --allow-system-changes
    -p tp:net:netif_receive_skb  \
    -p tp:net:net_dev_start_xmit \
    -p kprobe:skb_scrub_packet
98 event(s) processed
```



Example

```
477245817184291 (1) [ping] 286951 [tp] net:net_dev_start_xmit

  ns 4026537203 if 489 (pint) 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0 id 64110
off 0 len 84 proto ICMP (1) type 8 code 0


477245817189008 (1) [ping] 286951 [k] skb_scrub_packet

  ns 4026537203 if 489 (pint) 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0 id 64110
off 0 len 84 proto ICMP (1) type 8 code 0


477245817193895 (1) [ping] 286951 [tp] net:netif_receive_skb

  ns 4026531840 if 488 (local_pint) 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0 id
64110 off 0 len 84 proto ICMP (1) type 8 code 0
```

Example

```
477245817259437 (1) [ping] 286951 [k] __nft_trace_packet

  ns 4026531840 if 487 (br_int) rxif 487 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0
id 64110 off 0 len 84 proto ICMP (1) type 8 code 0

  table firewalld (58) chain mangle_PREROUTING (1) accept (policy)

  ct_state ESTABLISHED status 0x19a icmp orig [192.168.10.10 > 172.10.10.20 type 8
code 0 id 24807] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 24807] zone 0


477245817271805 (1) [ping] 286951 [k] __nft_trace_packet

  ns 4026531840 if 487 (br_int) rxif 487 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0
id 64110 off 0 len 84 proto ICMP (1) type 8 code 0

  table firewalld (58) chain filter_PREROUTING (13) accept (policy)

  ct_state ESTABLISHED status 0x19a icmp orig [192.168.10.10 > 172.10.10.20 type 8
code 0 id 24807] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 24807] zone 0
```

Filtering

Filtering

- Avoid ringbuf exhaustion
- Narrowing down is simply more convenient

Two ways of filtering:

- Packet based one
- Metadata based one

Packet based filtering

How it works:

- pcap-filter expression compiled into cBPF
- cBPF filter is then converted into eBPF
  - bpf_convert_filter() in net/core/filter.c
  - Packet loads through helper calls (BPF_FUNC_probe_read_kernel)

With some limitations:

- No custom BPF bytecode
- No bpf extensions

Packet based filtering

- The resulting eBPF instructions:
  - Initially used to freplace a stub
  - Problem: older kernels (no pointers support in global function args)
    - fentry evaluation + non straightforward workaround
  - TL;DR: ***`libbpf_register_prog_handler()/prog_prepare_load_fn()`***

```
asm volatile (
        "call %[filter];"
```

Meta data based filtering

Match against skb and inner data structures fields.

- ***sk_buff.mark == 0x11***
  - *skb->mark == 0x11*
- ***sk_buff.mark:0xff > 0***
  - *skb->mark & 0xff > 0*
- ***sk_buff.pkt_type == 3***
  - *skb->pkt_type == PACKET_OTHERHOST*
- ***sk_buff._nfct:~0x7:nf_conn.mark***
  - *((struct nf_conn *)(skb->_nfct & NFCT_PTRMASK))->mark != 0*
- ***sk_buff.dev.name == "eno1"***
  - *strncmp(skb->dev->name, name, IFNAMSIZ);*

Meta data based filtering

Sk_buff._nfct:~0x7:nf_conn.mark

```
((struct nf_conn *)(skb->_nfct & NFCT_PTRMASK))->mark != 0
```

Userspace

- Gets the BTF info for sk_buff
- Retrieves the offset of _nfct
- Emits a “load ptr” storing the mask
- Retrieves the BTF for nf_conn
  - Calculates the offset of mark
- Emits the load operation for `sizeof(mark)`
- Creates an entry for the RHS

eBPF

- Gets the informations on the rhs
- For the _nfct load_ptr calls

  `bpf_probe_read_kernel(dst, size, base + off)`
  - Updates the base address (& mask)
- For the non-ptr_load simply calls

  `bpf_probe_read_kernel(,, base + off)` and compare against the RHS

Example #3

```
$ retis collect -c skb,nft,ct –skb-sections dev,ns --allow-system-changes
    -f "icmp and ip src 192.168.10.10" \
    -p tp:net:netif_receive_skb        \
    -p tp:net:net_dev_start_xmit       \
    -p kprobe:skb_scrub_packet
12 event(s) processed
```

Example

```
[...]
477245817259437 (1) [ping] 286951 [k] __nft_trace_packet

  ns 4026531840 if 487 (br_int) rxif 487 192.168.10.10 > 172.10.10.20 ttl 64 tos 0x0
id 64110 off 0 len 84 proto ICMP (1) type 8 code 0

  table firewalld (58) chain mangle_PREROUTING (1) accept (policy)

  ct_state ESTABLISHED status 0x19a icmp orig [192.168.10.10 > 172.10.10.20 type 8
code 0 id 24807] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 24807] zone 0

[...]
```

Filtering - What's next?

Packets:

- Meaningful BPF extensions
  - vlan

Meta filter:

- Boolean expressions (series under review):
  - ```
    sk_buff.pkt_type == 0x0 ||
    (sk_buff.mark == 0x100 && sk_buff.cloned == 0)
    ```
- Extend this to non-skb types (including retvals)
    ```

Packets mutate

A need to track packets in the stack

(after matching a filter once)

- Packets can be modified (NAT, encapsulation, etc); might not match filters anymore
- Packets can be cloned or copied: the same initial packet can take different paths; filtering will match both
- But also different packets will likely be reported interleaved when probing more than one function / tracepoint; hard to follow

All right, but what's a packet in Linux?

- Metadata: ***struct sk_buff*** aka ~ Linux's representation of a packet
- Data: in-memory buffer pointed by ***sk_buff->head*** aka ~ what was / will be seen on the wire
- We could even consider messages in between the applications and ***sk_buff*** or XDP — not covered here

What to track? Either the metadata or the data addresses; some things to consider:

- Memory can be reused (once packet / metadata is freed)
- Operations can change the addresses (e.g. head reallocation)
- Different instances of metadata can point to the same data (e.g. clones)
- Packets can be merged or combined
- And some corner cases (e.g. valid "UaF" of an skb in a tracepoint https://lore.kernel.org/lkml/4DE877E1.7000606@jp.fujitsu.com/T/)

What to actually do?

No solution (tracking the metadata or data) works without any trick or handling corner cases. In Retis we track the data itself:

- Less corner cases, in our experience
- Tracks clones out of the box

We report two fields to identify packets:

- Unique identifier ("tracking id"): ***original_skb_head << 64 | initial_timestamp***
- ***sk_buff*** address; helps to distinguish between clones
- E.g. ***#57d008c23d9ffff93bc8e6a6580 (skb ffff93bc8fe2f700)***

See
https://github.com/retis-org/retis/blob/main/retis/src/core/tracking/skb_tracking.rs

What's next?

Associating tracking data to the packets directly would simplify the logic greatly: we could only attach an id the first time a packet is seen and ~ that's it.

- Should be doable from eBPF probes (set + get)
- Would be easier if it survives clones and copies; but not a blocker

Rich packet metadata?
https://netdevconf.info/0x19/sessions/talk/traits-rich-packet-metadata.html

Some open questions

- Should tagging packets be a standalone project? Could be used by different observability & debugging projects (and get the same id).
- Should generating & attaching an id part of the kernel (e.g. an eBPF kfunc)?

Example #3

```
$ retis collect -c skb,nft,ct,skb-tracking --skb-sections dev,ns
--allow-system-changes -o -print \
    -f "icmp and ip src 192.168.10.10" \
    -p tp:net:netif_receive_skb        \
    -p tp:net:net_dev_start_xmit       \
    -p kprobe:skb_scrub_packet

479029505138309 [...] #189cfc6893f06ffff908834850780 (skb ffff90887f433800)
  ns 4026537203 (pint) 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0
[...]
```

* manipulated output

Example

```
$ retis sort

479029505138309 (1) [ping] 331833 [tp] net:net_dev_start_xmit [...] n 0
  ns 4026537203 (pint) 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0

  ↳ 479029505149933 (1) [ping] 331833 [k] skb_scrub_packet [...] n 1
     ns 4026537203 (pint) 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0

  ↳ 479029505157992 (1) [ping] 331833 [tp] net:netif_receive_skb [...] n 2
     ns 4026531840 (local_pint) 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0


479029505492711 (6) [handler739] 3462839/3854908 [tp] net:netif_receive_skb n 0
  ns 4026531840 (br_int) 192.168.10.10 > 172.10.10.20 ttl 64 ICMP (1) type 8 code 0

  ↳ 479029505518657 (6) [handler739] 3462839/3854908 [k] __nft_trace_packet n 1
     [...]
```

* manipulated output

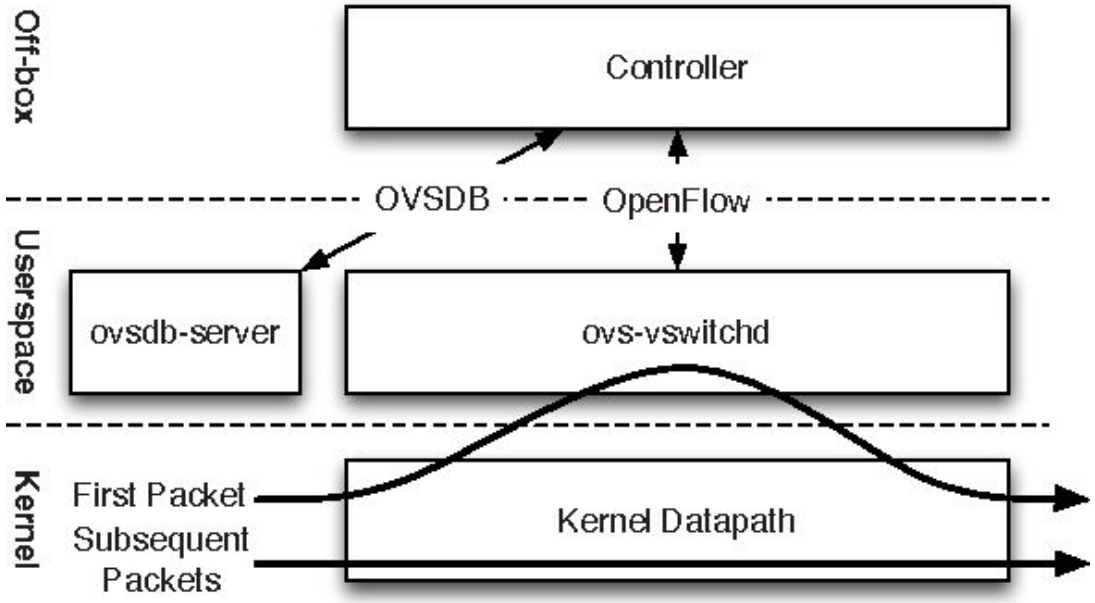
Example

```
  ↳ 479029505565036 (6) [handler739] 3462839/3854908 [k] __nft_trace_packet n 6
     ns 4026531840 (br_ext) rxif 487 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0
     table example_nat (55) chain postrouting (1) handle 14 accept
     ct_state NEW status 0x190 icmp orig [192.168.10.10 > 172.10.10.20 type 8 code 0 id
4153] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 4153] zone 0

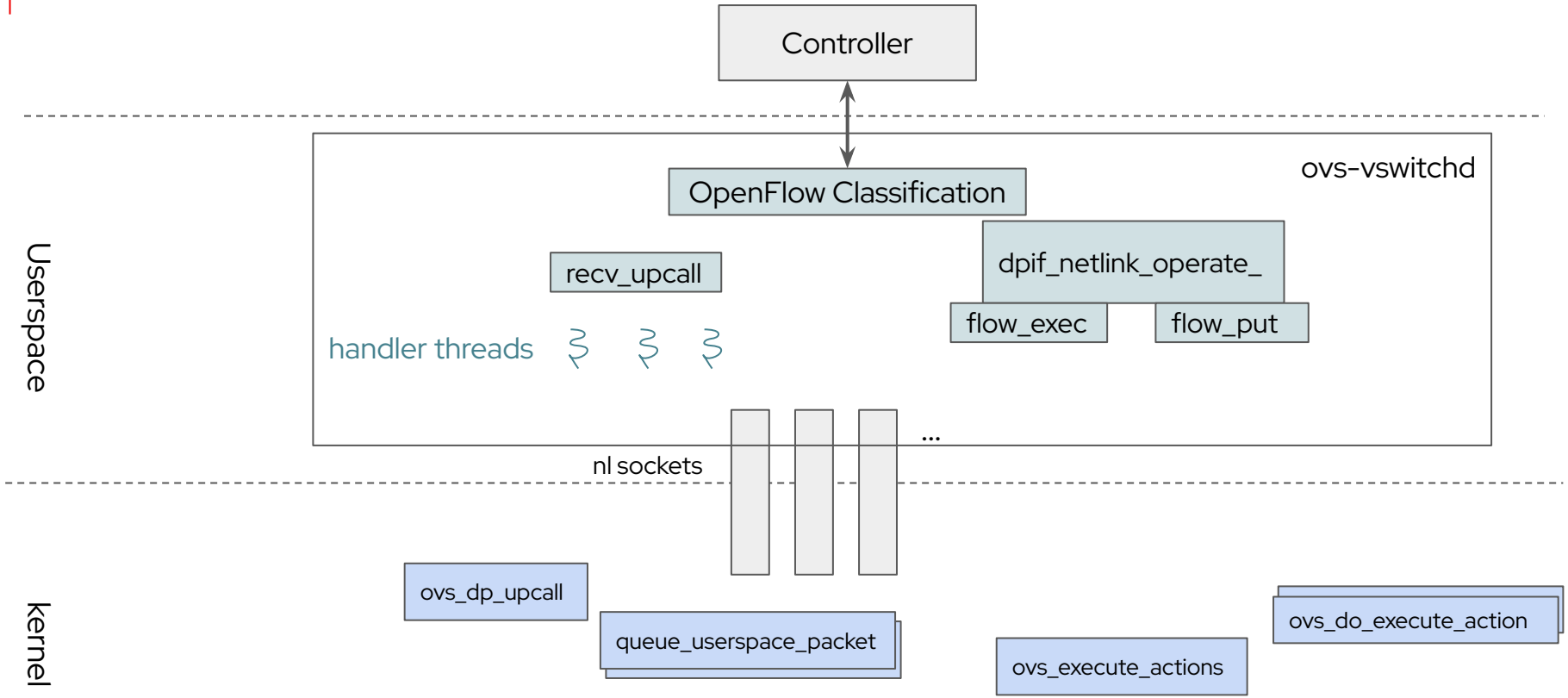
  ↳ 479029505570289 (6) [handler739] 3462839/3854908 [tp] net:net_dev_start_xmit n 7
     ns 4026531840 (br_ext) rxif 487 172.10.10.1 > 172.10.10.20 ICMP (1) type 8 code 0
     ct_state NEW status 0x198 icmp orig [192.168.10.10 > 172.10.10.20 type 8 code 0 id
4153] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 4153] zone 0
```

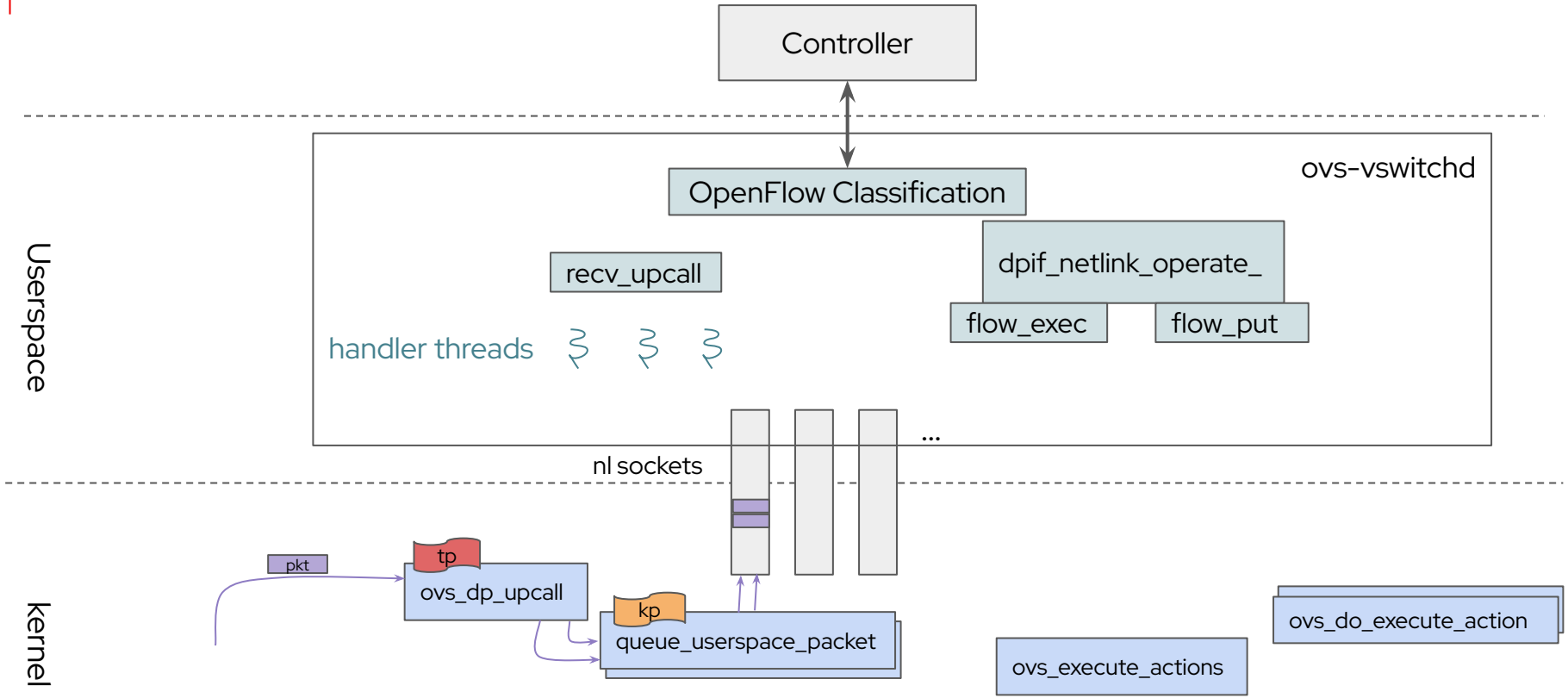
* manipulated output

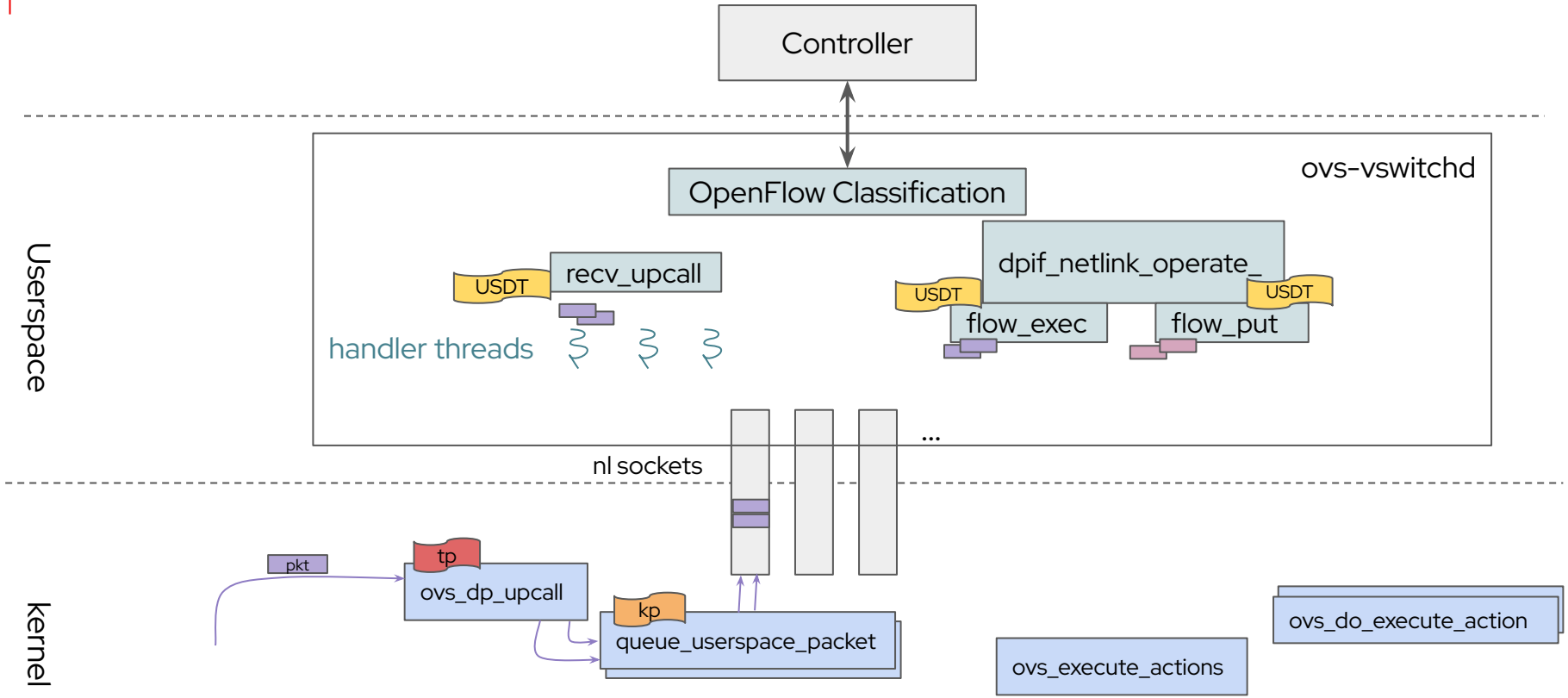
Packets sometimes go through userspace: OVS

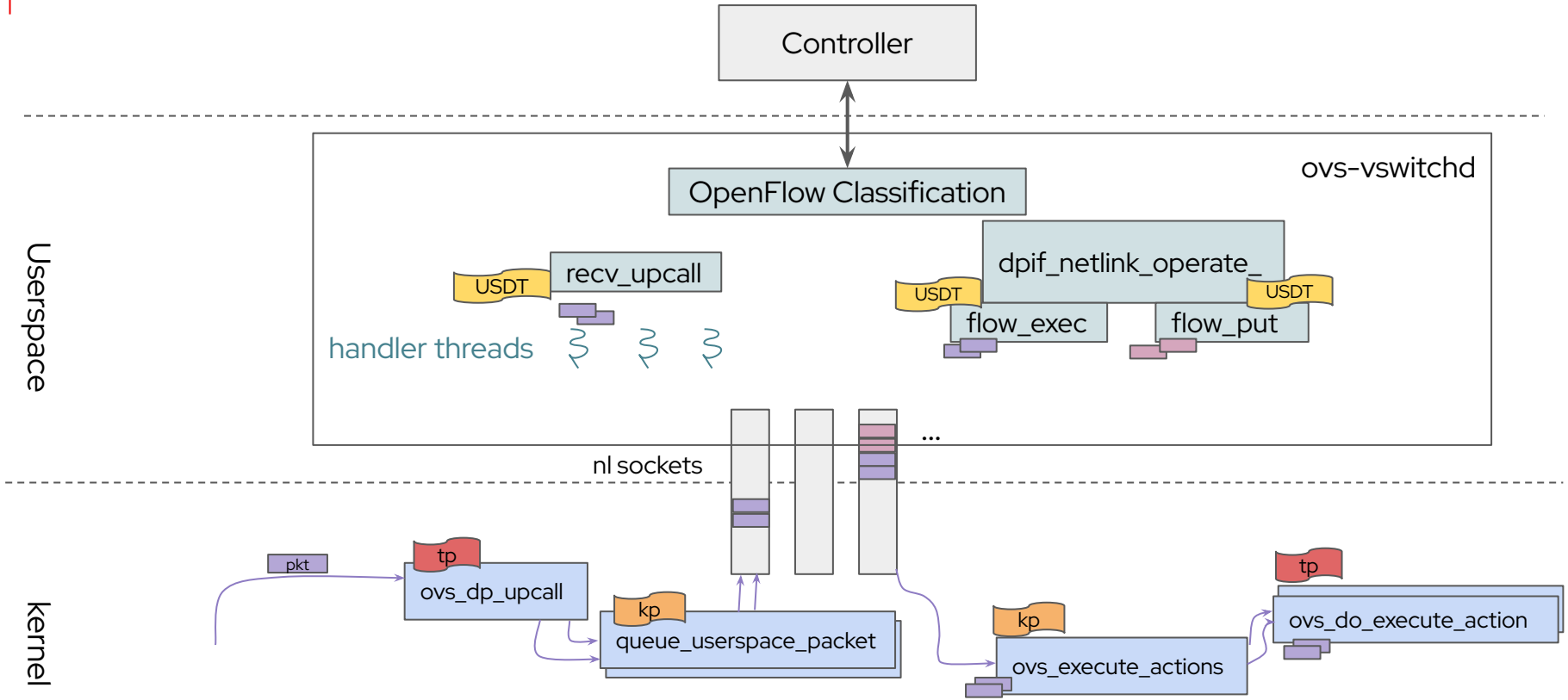


Source: The Design and Implementation of Open vSwitch
(https://www.semanticscholar.org/paper/The-Design-and-Implementation-of-Open-vSwitch/00cf4b13a1bd202ccebe1e7bd0587f11e98ec3d6)


Controller
ovs-vswitchd
OpenFlow Classification
recv_upcall
dpif_netlink_operate_
flow_exec
flow_put
handler threads
...
nl sockets
Userspace
kernel
ovs_dp_upcall
queue_userspace_packet
ovs_execute_actions
ovs_do_execute_action


Controller
OpenFlow Classification
ovs-vswitchd
recv_upcall
dpif_netlink_operate_
flow_exec
flow_put
handler threads
...
nl sockets
Userspace
kernel
pkt
tp
ovs_dp_upcall
kp
queue_userspace_packet
ovs_execute_actions
ovs_do_execute_action


Controller
Userspace
ovs-vswitchd
OpenFlow Classification
USDT
recv_upcall
handler threads
dpif_netlink_operate_
USDT
USDT
flow_exec
flow_put
...
nl sockets
kernel
pkt
tp
ovs_dp_upcall
kp
queue_userspace_packet
ovs_execute_actions
ovs_do_execute_action


Controller
OpenFlow Classification
ovs-vswitchd
Userspace
USDT
recv_upcall
handler threads
dpif_netlink_operate_
USDT
USDT
flow_exec
flow_put
nl sockets
...
kernel
pkt
tp
ovs_dp_upcall
kp
queue_userspace_packet
kp
ovs_execute_actions
tp
ovs_do_execute_action

Tracking: kernel -> userspace -> kernel?

- Use netlink seq number?
  - re-injection socket != upcall socket
- Use a custom OVS-specific “upcall id”?
- Rich packet metadata (a.k.a traits)? OVS should persist it across upcalls *anyways*
  - Depend on external entities adding tracking data or should OVS add its own?
  - How to deal with trait re-injection?
- What we did?
  - Use packet data hash + knowledge of OVS internals
  - Stitch things together at post-processing time (*retis sort*)

USDT Probes

- No BTF, no CO-RE ->very little flexibility
- Idea: Debuginfo (dwarf) -> (pahole) -> BTF -> CO-RE?
  - Mixed kernel & user BTF
  - pahole does not fully support .dwz compressed files yet
  - Replace pahole with gimli + (??)?

Example #4

```
$ retis collect -c skb,nft,ct,skb-tracking,ovs --ovs-track
--allow-system-changes -o \
    -f "icmp and ip src 192.168.10.10" \
    -p tp:net:netif_receive_skb        \
    -p tp:net:net_dev_start_xmit       \
    -p kprobe:skb_scrub_packet
$ retis sort
```

Example

```
480291159523957 (10) [ping] [tp] net:net_dev_start_xmit (pint) n 0
  ns 4026537203 192.168.10.10 > 172.10.10.20  ICMP (1) type 8 code 0

  ↳ 480291159544018 (10) [ping] [k] skb_scrub_packet (pint) n 1
  ↳ 480291159561831 (10) [ping] [tp] net:netif_receive_skb (local_pint) n 2
  ↳ 480291159585191 (10) [ping] [tp] openvswitch:ovs_dp_upcall n 3
  ↳ 480291159621671 (10) [ping] [kr] queue_userspace_packet n 4
  ↳ 480291159633626 (10) [ping] [kr] ovs_dp_upcall n 5
  ↳ 480291160059643 (0) [handler748] [u] dpif_recv:recv_upcall n 6
  ↳ 480291160153037 (0) [handler748] [u] dpif_netlink_operate__:op_flow_put n 7
  ↳ 480291160156186 (0) [handler748] [u] dpif_netlink_operate__:op_flow_execute n 8
  ↳ 480291160191550 (0) [handler748] [tp] openvswitch:ovs_do_execute_action n 9
     exec oport 2
  ↳ 480291160199091 (0) [handler748] [tp] net:netif_receive_skb (br_int) n 10
  ↳ 480291160228122 (0) [handler748] [k] __nft_trace_packet n 11
```

* manipulated output

Example

```
  ↳ 480291160270746 (0) [handler748] [k] __nft_trace_packet n 12
[...]
  ↳ 480291160287197 (0) [handler748] [k] __nft_trace_packet n 16
     (br_ext) 192.168.10.10 > 172.10.10.20 ICMP (1) type 8 code 0
     table example_nat (55) chain postrouting (1) handle 14 accept
     ct_state NEW status 0x190 icmp orig [192.168.10.10 > 172.10.10.20 type 8 code 0 id
36371] reply [172.10.10.20 > 172.10.10.1 type 0 code 0 id 36371] zone 0 mark 0
  ↳ 480291160294623 (0) [handler748] [tp] net:net_dev_start_xmit n 17
     (br_ext) rxif 487 172.10.10.1 > 172.10.10.20 ICMP (1) type 8 code 0
  ↳ 480291160301227 (0) [handler748] [tp] net:net_dev_start_xmit n 18
  ↳ 480291160303101 (0) [handler748] [k] skb_scrub_packet (br_ext)
  ↳ 480291160306705 (0) [handler748] [tp] net:netif_receive_skb (pext)
```

* manipulated output

Conclusion

Some extra features

- Custom analysis using **python bindings**
- Help users that are not as familiar with the stack
  - **Profiles** (easy-to-share yaml files describing cli args)
  - Reasonable defaults
  - **Auto-probe** (kernel stack -> probes)

... and many more

Open questions / Recap

- Long term availability of ***prog_prepare_load_fn***?
- Rich packet metadata for packet tracking? Including OVS upcalls?
- Sharing logic for tracking packets (new project, in-kernel, etc)?
- CO-RE for USDT probes?

Thank you

Repo: https://github.com/retis-org/retis/

Docs: https://retis.readthedocs.io/en/stable/