XDP gaining access to NIC hardware hints via BTF

Jesper Dangaard Brouer

Sr. Principal Kernel Engineer

Red Hat Inc.

Linux Plumbers Conference
September 2022

What are XDP-hints

XDP-hints dates back to NetDevConf Nov 2017 (by PJ Waskiewicz)

- Purpose: Let XDP access HW offload hints

Basic idea:

- Provide or extract (from descriptor) NIC hardware offload hints
- Store info in XDP metadata area (located before pkt header)

XDP metadata area avail since Sep 2017 (by Daniel Borkmann)

- Space is limited (currently 32 bytes)

Main reason XDP-hints work stalled

- No consensus on layout of XDP metadata
- BTF was not ready at that time (BPF Type Format)

What are traditional hardware offload hints?

NIC hardware provides offload hints in RX (and TX) descriptors

- The netstack SKB packet data-struct stores+uses these

RX descriptors can e.g. provide:

- RX-checksum validation, RX-hash value, RX-timestamp
- RX-VLAN provides VLAN ID/tag non-inline

TX descriptors can e.g. ask hardware to perform actions:

- TX-checksum: Ask hardware to compute checksums on transmission
- TX-VLAN: Ask hardware to insert VLAN tag
- Advanced: TX-timestamp HW stores TX-time and feeds back on completion
- Advanced: TX-LaunchTime ask HW to send packet at specific time in future

XDP-hints layout defined via BTF layout

My proposal: Use BTF to define the layout of XDP metadata

- Each NIC driver can choose its own BTF layout
- Slightly challenging requirement:
  - NIC driver can change layout per pkt (e.g timestamp only in PTP pkts)

Open question:

- Will BTF be a good fit for this use-case?

Next slides: Explaining BTF technical details

Assume LPC crowd knows what BTF is

- Slides are here primarily for people downloading these later

Focused on getting mind share on:

- What are BTF IDs ?
  - Especially: BTF object vs. type IDs

Introducing BTF - BPF Type Format

BTF compact Type Format (based on compiler's DWARF debug type info)

- Great blogpost by Andrii Nakryiko
  - 124MB of DWARF data compressed to 1.5MB compact BTF type data
- Suitable to be included in Linux kernel image by default
  - See file `/sys/kernel/btf/vmlinux` avail in most distro kernels
- Kernel's runtime data structures have become self-describing via BTF

```
# bpftool btf dump file /sys/kernel/btf/vmlinux format c
```

More components: CO-RE + BTF + libbpf

Blogpost on BPF CO-RE (Compile Once – Run Everywhere) (Andrii Nakryiko)

- Explains how BTF is one piece of the puzzle
- BPF ELF object files are made portable across kernel versions via CO-RE
- LLVM compiler emits BTF relocations (for BPF code accessing struct fields)

BPF-prog (binary ELF object) loader libbpf combines pieces

- Tailor BPF-prog code to a particular running kernel
- Looks at BPF-prog recorded BTF type and relocation information
  - matches them to BTF information provided by running kernel
  - updates necessary offsets and other relocatable data
- Kernel struct can change layout, iff member name+size stays same

Code-Example: Partial struct + runtime BTF-id

BPF-prog can define partial struct with few members

- libbpf matches + “removes” triple-underscore after real struct name
- preserve_access_index will be matched against kernel data-structure

```
struct sk_buff___local {
        __u32 hash;
} __attribute__((preserve_access_index));

SEC("kprobe/udp_send_skb.isra.0")
int BPF_KPROBE(udp_send_skb, struct sk_buff___local *skb)
{
        __u32 h; __u32 btf_id;
        BPF_CORE_READ_INTO(&h, skb, hash); /* skb->hash */
        btf_id = bpf_core_type_id_kernel(struct sk_buff___local); /* libbpf load-time lookup */
        bpf_printk("skb->hash=0x%x btf_id(skb)=%d", h, btf_id);
}
```

Notice: Can get BTF type id for sk_buff used by running kernel

BTF type IDs and their usage

BTF system has type IDs to refer to each-other (in compressed format)

- Zero is not a valid BTF ID and numbering (usually) starts from one
  - Userspace can dump and see numbering via `bpftool btf dump file`

Kernel's BTF data files are located in `/sys/kernel/btf/` (modules since v5.11)

- Main file vmlinux contains every type compiled into kernel
- All module files offset ID numbering to start at last vmlinux ID
  - Allows modules to reference vmlinux type IDs (for compression)

Userspace BPF-prog ELF-object files also contains BTF sections

- This is known as local BTF and numbering starts at one
- BPF-prog can query own local BTF id via: `bpf_core_type_id_local()`

Q: Can we identify BTF layout via the BTF ID?

Issue: BTF type IDs are not unique (32-bit)

- But they are unique within one BTF object

BTF (objects) loaded into kernel also have an (BTF) ID (32-bit)

- “vmlinux” gets ID 1
- modules gets IDs assigned on loading
- same for user loaded BTF objects

Construct unique: Full BTF ID (64-bit)

- via combining: BTF object and type ID

Back to XDP-hints

Back to XDP-hints and XDP metadata area

XDP metadata requirements

XDP metadata area has some properties

- Grows "backwards" from where packets starts
- Must be 4 byte aligned
- Limited size (currently) 32 bytes

BPF-prog can expand/grow area via helper: `bpf_xdp_adjust_meta`

- pkt-data pointers are invalidated after calling this
- Verifier requires boundary checks to access metadata area

Common gotcha: Compiler likes to pad C-struct ending

- Avoid/fix via: `__attribute__((packed))`

Expected users of the XDP-hints

Users/consumers of XDP-hints in BTF layout

- BPF-progs first obvious consumer (either XDP or TC hooks)
- XDP to SKB conversion (in veth and cpumap) for traditional HW offloads
  - e.g. RX-hash, RX-checksum, VLAN, RX-timestamp
  - Can potentially simplify NIC drivers significantly
- Chained BPF-progs can communicate state via metadata
- AF_XDP can consume BTF info in userspace to decode metadata area

Motivation for XDP to SKB conversion

Moonshot: NIC drivers without SKB knowledge

- End-goal with XDP to SKB conversion
- Make it possible to write NIC drivers Ethernet L2 “only”

Goal: Avoids taking the SKB “socket” overhead at driver level

- When early netstack layer 2/3 processing xdp_frames
  - Possible to speedup Linux bridging and routing

Hardware motivation and considerations

Goal: Hardware should produce XDP-hints

- Possible for HW as DMA area next to metadata

Consider defining Endianess: Big vs Little endian

- In XDP-hints struct layout
- Given BTF is flexible, can be added later when HW appears

XDP-hints exploring solutions using BTF

Design not set in stone yet

- Upstream interaction will likely change solution anyhow

Explaining current approach

Step#1: Decouple with btf_id in metadata

Place full “btf_id” value inside metadata area, as last member

- last member: due to “grows” backwards, important for AF_XDP decoding
- Extend `xdp_buff` + `xdp_frame` (+AF_XDP) with flags that BTF is “enabled”
  - Notice: Full BTF ID identify which module via BTF object ID

This achieves decoupling via btf_full_id - no locked/fixed XDP struct

- Pros: Easy to handle different layout per pkt
  - as BPF-prog (or AF_XDP) can multiplex on btf_id’s known to “them”
- Cons: XDP to SKB conversion harder
  - Would need table lookup for each compat layouts

Step#2: Extend with common struct

Create `xdp_hints_common` struct with netstack known hints

- Still place “btf_full_id” value inside metadata area, as last member
- Extend `xdp_buff` with flag: ’compat with common hints’
- Helps XDP to SKB use-case

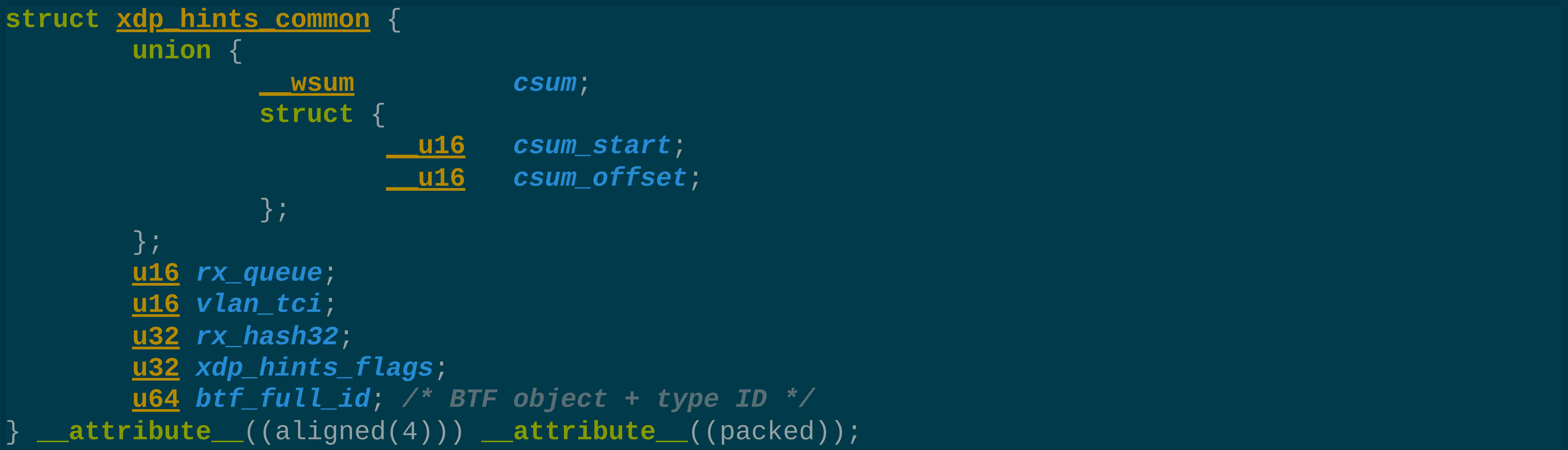
Userspace MUST not consider this common struct UAPI

- Kernel can change this anytime
- Userspace MUST use BTF info to decode layout

This is proposal in RFC v2 patchset

- [RFCv2] XDP-hints: XDP gaining access to HW offload hints via BTF

Layout of xdp_hints_common



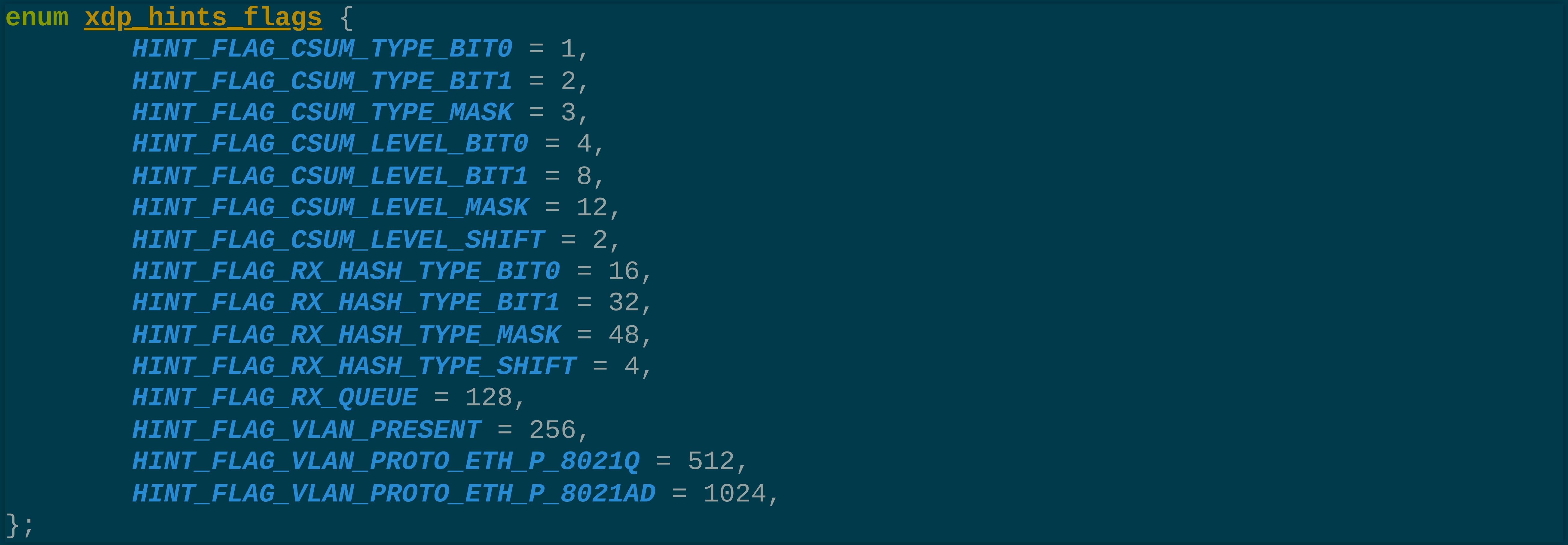
```
struct xdp_hints_common {
        union {
                __wsum          csum;
                struct {
                        __u16   csum_start;
                        __u16   csum_offset;
                };
        };
        u16 rx_queue;
        u16 vlan_tci;
        u32 rx_hash32;
        u32 xdp_hints_flags;
        u64 btf_full_id; /* BTF object + type ID */
} __attribute__((aligned(4))) __attribute__((packed));
```

Member `xdp_hints_flags` is further ’described’ via

- BTF type `enum xdp_hints_flags`

BTF type enum xdp_hints_flags

Not UAPI: BPF-prog + userspace MUST decode via BTF



```
enum xdp_hints_flags {
        HINT_FLAG_CSUM_TYPE_BIT0 = 1,
        HINT_FLAG_CSUM_TYPE_BIT1 = 2,
        HINT_FLAG_CSUM_TYPE_MASK = 3,
        HINT_FLAG_CSUM_LEVEL_BIT0 = 4,
        HINT_FLAG_CSUM_LEVEL_BIT1 = 8,
        HINT_FLAG_CSUM_LEVEL_MASK = 12,
        HINT_FLAG_CSUM_LEVEL_SHIFT = 2,
        HINT_FLAG_RX_HASH_TYPE_BIT0 = 16,
        HINT_FLAG_RX_HASH_TYPE_BIT1 = 32,
        HINT_FLAG_RX_HASH_TYPE_MASK = 48,
        HINT_FLAG_RX_HASH_TYPE_SHIFT = 4,
        HINT_FLAG_RX_QUEUE = 128,
        HINT_FLAG_VLAN_PRESENT = 256,
        HINT_FLAG_VLAN_PROTO_ETH_P_8021Q = 512,
        HINT_FLAG_VLAN_PROTO_ETH_P_8021AD = 1024,
};
```

BTF type enum xdp_hints_csum_type

The HINT_FLAG_CSUM_TYPE’s are mapped to SKB usage

- via BTF defined enum - not UAPI

```
enum xdp_hints_csum_type {
        HINT_CHECKSUM_NONE          = CHECKSUM_NONE,
        HINT_CHECKSUM_UNNECESSARY = CHECKSUM_UNNECESSARY,
        HINT_CHECKSUM_COMPLETE     = CHECKSUM_COMPLETE,
        HINT_CHECKSUM_PARTIAL      = CHECKSUM_PARTIAL,
};
```

Driver specific struct

Example: Driver specific struct

- Simply include common struct as last member

```
struct xdp_hints_i40e {
        struct i40e_rx_ptype_decoded i40e_hash_ptype;
        struct xdp_hints_common common;
};
```

Driver devel must make sure `btf_full_id` is last member

- Watch out for C-compiler padding
- And comply with metadata 4 byte alignment rules

What BTF layout does a driver provide?

How to solve “exporting” available BTF-layouts

- per NIC driver

Is a new really UAPI needed?!?

- Just use BTF ???

What BTF layout does this driver provide?

How does userspace (and libbpf) know:

- What BTF layout does this driver provide?

Proposal: Struct naming-convention for `struct xdp_hints_*`

- Could be way for drivers to “export” available BTF-layouts?

New UAPI is not really needed:

- Remember: BTF info avail via `/sys/kernel/btf/`
  - both for vmlinux and modules
- libbpf parses and resolves relocations via these
- AF_XDP userspace can also decode BTF

Proposal: Encapsulating C-code union?

Each NIC driver could have a `union` named xdp_hints_union

- Structs added to union, means driver may use this BTF layout
- Notice: Union “sub” structs automatically gets own BTF IDs
- Essentially: Way to describe/support NIC using layouts per packet

Complications: metadata grows backwards

- Padding needed if union should match memory layout
  - Cons: Union padding quickly gets “ugly” in C-code
  - Pros: Easier for driver C-code with one type for metadata area

Example xdp_hints_union for driver i40e

```c
/* xdp_hints_union defines xdp_hints_* structs available in this driver.
 * As metadata grows backwards structure are padded to align.
 */
union xdp_hints_union {
        struct xdp_hints_i40e_timestamp i40e_ts;
        struct {
                u64 pad1_ts;
                struct xdp_hints_i40e i40e;
        };
        struct {
                u64 pad2_ts;
                u32 pad3_i40e;
                struct xdp_hints_common common;
        };
} __aligned(4) __attribute__((packed));
```

The actual C-code doesn’t look that ugly, right?

- and fits a single slide with room to spare

Future work

Mostly covered RX-side:

- Future work TX-side: 'ask hardware to perform actions'
- Also TX-completion event can return HW hints, e.g. wire TX-time

Help userspace developers decode BTF

- Code more examples and perhaps make lib
- Listing of avail xdp_hints_* (via btftool?)

End: Questions?

Resources:

- XDP-project - GitHub.com/xdp-project
  - Get an easy start with xdp-project/bpf-examples
- XDP-hints mailing list: xdp-hints @ xdp-project.net
  - https://lists.xdp-project.net/