XDP hints via BPF Type Format (BTF) system

Jesper Dangaard Brouer
Senior Principal Kernel Engineer

Lund Linux Con
April 2022

Reminder: What is BPF ?

From: https://ebpf.io/what-is-ebpf

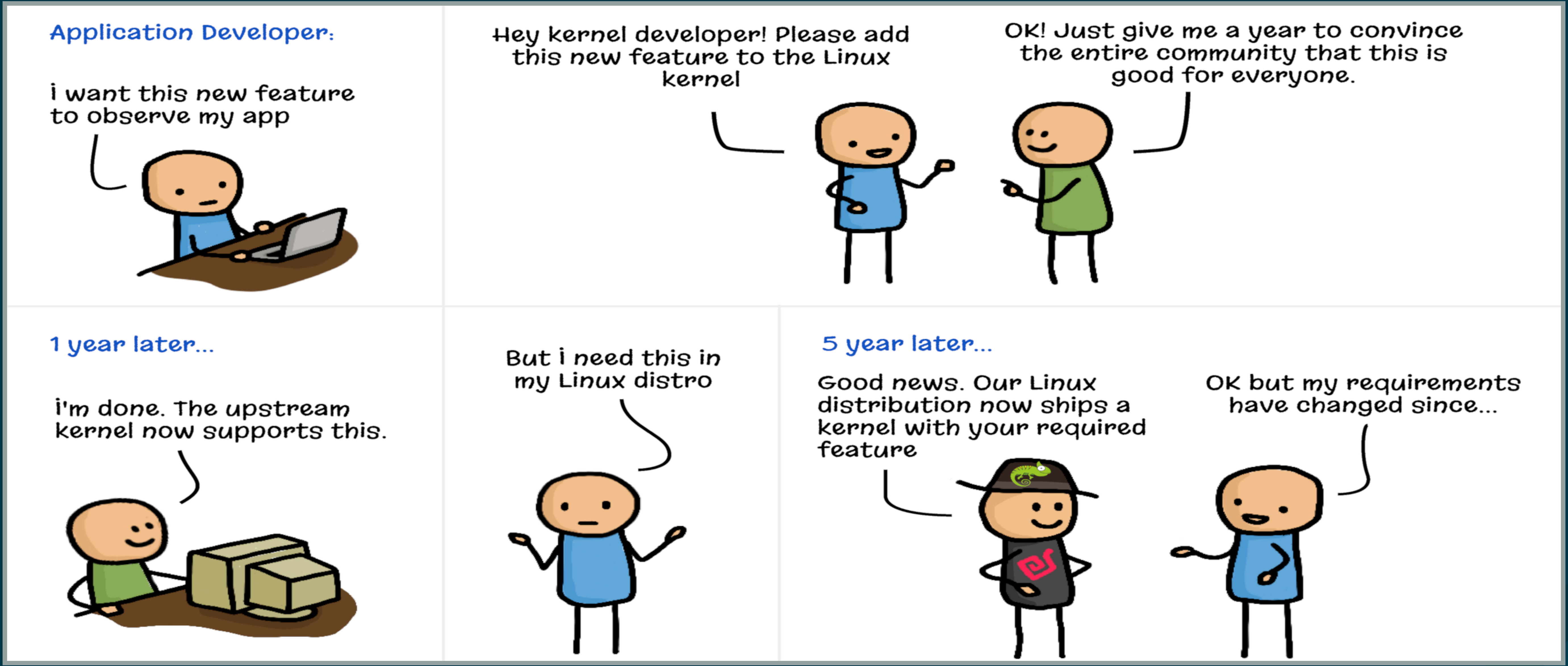
> *BPF is a revolutionary technology that can run sandboxed programs in the Linux kernel without changing kernel source code or loading a kernel module*

BPF is a technology name: no longer an acronym

Rate of innovation at the operating system level: Traditionally slow

- BPF enables things at OS-level that were not possible before
- BPF will radically increase rate of innovation

Traditional Kernel development process


Application Developer:
I want this new feature to observe my app
Hey kernel developer! Please add this new feature to the Linux kernel
OK! Just give me a year to convince the entire community that this is good for everyone.
1 year later...
I'm done. The upstream kernel now supports this.
But I need this in my Linux distro
5 year later...
Good news. Our Linux distribution now ships a kernel with your required feature
OK but my requirements have changed since...

BPF development process

Application Developer:
i want this new feature to observe my app
eBPF Developer:
OK! The kernel can't do this so let me quickly solve this with eBPF.
A couple of days later...
Here is a release of our eBPF project that has this feature now. BTW, you don't have to reboot your machine.

Reminder: What is XDP?

XDP (eXpress Data Path) is a Linux in-kernel fast-path

- Programmable layer in-front of traditional network stack
  - Read, modify, drop, redirect or pass
  - For L2-L3 use-cases: seeing x10 performance improvements!
- Avoiding memory allocations
  - No SKB allocations and no-init (SKB zeroes 4 cache-lines per pkt)
- Adaptive bulk processing of frames
- Very early access to frame (in driver code after DMA sync)
- Ability to skip (large parts) of kernel code
  - Evolve XDP via BPF-helpers
- Pitfall: Lost traditional HW offloads (e.g. RX-hash, checksum etc.)

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

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

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

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);
        bpf_printk("skb->hash=0x%x btf_id(skb)=%d", h, btf_id);
}
```

Notice: Can get btf_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 module 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()`

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”

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

- Next step: Speedups Linux bridging and routing (with xdp_frame)
  - Meaning: Normal Linux netstack get speedup for routing use-cases

Hardware motivation and considerations

Goal: Hardware should produce XDP-hints

- Easy 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 fully done yet

- Upstream interaction will likely change solution anyhow

Next slides: Proposed solutions with pros and cons

Solution(A): Internal kernel focus

Kernel-side: Could extend `xdp_buff` + `xdp_frame` with “btf_id” or ptr

- Pros: Gives BPF-prog access to reading `ctx->btf_id`
  - Cons: AF_XDP cannot read this ctx->btf_id
  - Unknown: Can chained BPF-progs update/write ctx->btf_id ?
- Pros: Kernel internally can (likely) store pointer to btf struct
  - Cons: This needs reference counting and race/lifetime handling
- Pros: XDP to SKB conversion (should be) easier to extract offloads
  - Unknown: Will it be harder to support different layout per pkt ???

Details: If dropping requirement layout per pkt

- Possible to store btf pointer in `xdp_rxq_info`

Solution(B): Decouple with btf_id in metadata

Place “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: Need 3 flags for BTF “origin” (vmlinux, module or local)
    - module resolved via `xdp_rxq_info` or `xdp_frame->dev_rx`

This achieves decoupling via btf_id as it becomes struct’s “version” number

- Pros: Easy to handle different layout per pkt
  - as BPF-prog (or AF_XDP) can multiplex on btf_ids known to “them”
- Cons: XDP to SKB conversion harder as kernel cannot trust btf_id
  - Solution: Add new BPF hook at XDP to SKB point (BPF prog builds skb)

Solution(C): Combined proposal

Still place “btf_id” value inside metadata area, as last member

- BUT is considered cached version from kernel stored pointer to btf
  - Cons: reference counting and lifetime handling still needed

Pros: XDP to SKB conversion can work

- via check if btf_id matches btf-ptr id before trusting BTF layout
- Details: For “lifetime” module BTF driver could disable this step on teardown

Pros: Chained BPF-progs works

- (last_member) btf_id becomes a communication channel
  - Unknown: How to communicate BTF “origin” (vmlinux, module or local)?

What BTF layout does a driver provide?

How to solve “exporting” available BTF-layouts

- per NIC driver

Is a new UAPI needed?

What BTF layout does this driver provide?

How does userspace (and libbpf) know:

- What BTF layout does this driver provide?

New UAPI might not be needed:

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

Struct naming-convention for xdp_hints

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

Proposal: Encapsulating C-code union?

Each NIC driver could have a `union` named xdp_hints_union

- Structss 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

Define “generic” xdp_hints common struct

Idea: Partly UAPI/kABI approach

Kernel (not module) `struct xdp_hints_common` (vmlinux BTF id)

- Should cover today’s known SKB offload hints
- Could have some defines as UAPI in bpf.h (e.g. flags hash-type, csum-type)

NIC drivers can let their xdp_hints include common struct as member

- Can extend with NIC specific hints by adding flag in xdp_buff/xdp_frame
    - That indicates layout is compatible with “xdp_hints_common”

Pros: Easier to implement XDP to SKB transition

- Cons: Goes against the fully dynamic BTF based layout

Metadata + BTF = communication channel

Using metadata area to communication state

- Create structure via BTF

Relevant for:

- Chained BPF-progs (between XDP to TC hooks, also BPF tail-calls)
- XDP-prog sending info to userspace AF_XDP sockets

Already: Works today!

Example use-case: XDP-prog to AF_XDP

Code: How to transfer info from XDP-prog to AF_XDP via BTF

- GitHub XDP-project: bpf-examples/ AF_XDP-interaction
- Use-case: XDP RX-timestamp for Real-Time TTEthernet sync (PCF frames)
- Changes layout per packet via btf_id as last member

Uses: local BTF info in BPF-prog ELF object

- Thus, no kernel extensions needed
- BPF-prog gets own local BTF id via: `bpf_core_type_id_local()`

Shows userspace C-code decoding BTF format

- Extracts offset + size for named struct members

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/

Extra slides

Traditional hint: RX-hash - implementation details

Kernel SKB: Hash value (only) 32-bit

- pkt_hash_types: PKT_HASH_TYPE_{NONE,L2,L3,L4} (bit skb->l4_hash)
- SKB bit (`skb->sw_hash`) if software computed hash

Hardware provides RSS-hash (standardised by Microsoft)

- Kernel drops info, RSS hashing type identify:
  - Tell us if this is IPv4: UDP or TCP
  - IPv6: UDP or TCP, and if extension headers are present

Traditional hint: VLAN both RX and TX - impl.

net_device feature flags for enabling VLANs offload hints:

- NETIF_F_HW_VLAN_{CTAG,STAG}_{RX,TX}_BIT
- C-tag = inner tag Customer-tag, S-tag = outer tag Service-provider-tag

Kernel SKB fields:

- vlan_present (1-bit), vlan_tci (16-bits), vlan_proto (BE 16-bit)

ethtool –{show-}features: rx-vlan-offload + tx-vlan-offload

Traditional hint: RX-checksum - impl. details

net_device feature flags avail for checksum capabilities

- e.g. NETIF_F_{HW,IP,IPV6}_CSUM

Kernel stores checksum type (in `skb->ip_summed` 2-bits)

- CHECKSUM_{NONE,UNNECESSARY,COMPLETE,PARTIAL}

CHECKSUM_COMPLETE fills skb->csum (union with csum_start+csum_offset)

CHECKSUM_PARTIAL needs skb_checksum_start_offset (skb->csum_start)

- Depends on `skb->csum_start` + `skb->csum_offset`

Encap: `skb->csum_level` consecutive checksums found in pkt

- Minus ones verified as CHECKSUM_UNNECESSARY
- Encap case also sets bit skb->encapsulation

Traditional hint: TX-checksum - impl. details

Requests net_device to update packet checksum fields

- SKB reusing types in `skb->ip_summed` (2-bits)
- features: NETIF_F_{IP,IPV6,HW}_CSUM

CHECKSUM_PARTIAL meaning at TX

- Do checksumming from `skb->csum_start` up to the end
- In pkt store checksum at offset `skb->csum_start + skb->csum_offset`