A practical introduction to XDP

Jesper Dangaard Brouer (Red Hat)
Andy Gospodarek (Broadcom)

Linux Plumbers Conference (LPC)
Vancouver, Nov 2018

What will you learn?

- Introduction to XDP and relationship to eBPF
- Foundational elements of XDP
- XDP program sample code
- Advanced XDP concepts
- Notes for driver developers

What is XDP?

- New, programmable layer in the kernel network stack
- Run-time programmable packet processing inside the kernel not kernel-bypass
- Programs are compiled to platform-independent eBPF bytecode
- Object files can be loaded on multiple kernels and architectures without recompiling

XDP design goals

Close the performance gap to kernel-bypass solutions

- Not a goal to be faster than kernel-bypass
- Operate directly on packet buffers (DPDK)
- Decrease number of instructions executed per packet

Operate on packets before being converted to SKBs

- Kernel Network stack built for socket-delivery use-case

Work in concert with existing network stack without kernel modifications

- Provide in-kernel alternative, that is more flexible
- Don’t steal the entire NIC!

XDP kernel hooks

Native Mode XDP

- Driver hook available just after DMA of buffer descriptor
- Process packets before SKB allocation - No waiting for memory allocation!
- Smallest number of instructions executed before running XDP program
- Driver modification required to use this mode

SKB or Generic Mode XDP

- XDP hook called from netif_receive_skb()
- Process packets after packet DMA and skb allocation completed
- Larger number of instructions executed before running XDP program
- /Driver indep

XDP relationship with eBPF

How is this connected?

Design: XDP: data-plane and control-plane

Data-plane: inside kernel, split into:

- Kernel-core: Fabric in charge of moving packets quickly
- In-kernel eBPF program:
  - Policy logic decide action
  - Read/write access to packet buffers

Control-plane: Userspace

- Userspace load eBPF program
- Can control program via changing eBPF maps
- Everything goes through BPF-syscall

XDP driver hook is executing eBPF bytecode

XDP puts no restrictions on how eBPF bytecode is generated or loaded

- XDP simply attaches eBPF file-descriptor handle to netdev

eBPF bytecode (and map-creation) all go-through BPF-syscall

- Provide hand-written eBPF instructions (not practical)
- Use LLVM+clang to generate eBPF bytecode in one of two ways
  - bcc-tools - compile eBPF each time program runs
  - libbpf - load ELF-object created by LLVM/clang

Code examples in this talk

This talk focus on approach used in $KERNEL_SRC/samples/bpf)

- Writing restricted-C code in foo_kern.c
  - eBPF code is restricted to protect kernel (not Turing complete)
- Compile to ELF object file foo_kern.o
- Load via libbpf (kernel tools/lib/bpf) as XDP data-plane
- Have userspace control-plane program foo_user.c via shared BPF-maps

Basic building blocks

What are the basic XDP building block you can use?

XDP actions and cooperation

eBPF program (in driver hook) return an action or verdict

- XDP_ DROP, XDP_ PASS, XDP_ TX, XDP_ ABORTED, XDP_ REDIRECT

How to cooperate with network stack

- Pop/push or modify headers: Change default rx_handler used by kernel
  - e.g. handle on-wire protocol unknown to running kernel
- Can propagate 32Bytes meta-data from XDP stage to network stack
  - TC (clsbpf) hook can use meta-data, e.g. set SKB mark
  - Pre-parse packet contents (XDP Hints) and store in this area
- Call eBPF helpers which are exported kernel functions
  - Helpers defined and documented in: include/uapi/linux/bpf.h

Evolving XDP via eBPF helpers

Think of XDP as a software offload layer for the kernel network stack

- Setup and use Linux kernel network stack
- Accelerate parts of it with XDP

IP routing application is great example:

- Let kernel manage route tables and perform neighbour lookups
- Access routing table from XDP program via eBPF helper: bpf_fib_lookup
- Rewrite packet headers if next-hop found, otherwise send packet to kernel
- This was covered in David Ahern's talk: Leveraging Kernel Tables with XDP

Similar concept could be extended to accelerate any kernel datapath

Add helpers instead of duplicating kernel data in eBPF maps!

Coding XDP programs

How do you code these XDP programs?

- Show me the code!!!

XDP restricted-C code example : Drop UDP

```
SEC("xdp_drop_UDP") /* section in ELF-binary and "program_by_title" in libbpf */
int xdp_prog_drop_all_UDP(struct xdp_md *ctx) /* "name" visible with bpftool */
{
        void *data_end = (void *)(long)ctx->data_end; void *data = (void *)(long)ctx->data;
        struct ethhdr *eth = data; u64 nh_off; u32 ipproto = 0;

        nh_off = sizeof(*eth); /* ETH_HLEN == 14 */
        if (data + nh_off > data_end) /* <-- Verifier use this boundry check */
                return XDP_ABORTED;

        if (eth->h_proto == htons(ETH_P_IP))
                ipproto = parse_ipv4(data, nh_off, data_end);
        if (ipproto == IPPROTO_UDP)
                return XDP_DROP;
        return XDP_PASS;
}
```

Simple XDP program that drop all IPv4 UDP packets

- Use struct ethhdr to access eth->h_proto
- Function call for parse_ipv4 (next slide)

Simple function call to read iph->protocol

```
static __always_inline
int parse_ipv4(void *data, u64 nh_off, void *data_end)
{
        struct iphdr *iph = data + nh_off;

        /* Note + 1 on pointer advance one iphdr struct size */
        if (iph + 1 > data_end) /* <-- Again verifier check our boundary checks */
                return 0;
        return iph->protocol;
}
```

Simple function call parse_ipv4 used in previous example

- Needs inlining as eBPF bytes code doesn't have function calls
- Again remember boundary checks, else verifier reject program

Coding with libbpf

libbpf: loading ELF-object code

Userspace program must call BPF-syscall to insert program info kernel

Luckily libbpf library written to help make this easier for developers

```
struct bpf_object *obj;
int prog_fd;

struct bpf_prog_load_attr prog_load_attr = {
        .prog_type = BPF_PROG_TYPE_XDP,
        .file      = "xdp1_kern.o",
};

if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
        return EXIT_FAILURE;
```

eBPF bytecode and map definitions from xdp1_kern.o are now ready to use and obj and prog_fd are set.

libbpf: ELF-object with multiple eBPF progs

```
struct bpf_object *obj;
int prog_fd;
struct bpf_prog_load_attr prog_load_attr = {
        .prog_type = BPF_PROG_TYPE_XDP,
        .file      = "xdp_udp_drop_kern.o",
};

if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd) == 0) {
        const char *prog_name = "xdp_drop_UDP"; /* ELF "SEC" name */
        struct bpf_program *prog;

        prog = bpf_object__find_program_by_title(obj, prog_name);
        prog_fd = bpf_program__fd(prog);
}
```

Possible to have several eBPF program in one object file

- libbpf can find program by section “title” name

libbpf: attaching XDP prog to ifindex

Now that a program is loaded (remember prog_fd set in the last snippet shown), attach it to a netdev

```
#include <"net/if.h"> /* if_nametoindex */
static __u32 xdp_flags = XDP_FLAGS_DRV_MODE /* or XDP_FLAGS_SKB_MODE */
static int ifindex = if_nametoindex("eth0");

if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
        printf("link set xdp fd failed\n");
        return EXIT_FAILURE;
}
```

If bpf_set_link_xdp_fd() is successful, the eBPF program in xdp1_kern.o is attached to eth0 and program runs each time a packet arrives on that interface.

Coding with eBPF maps

Accessing eBPF map from within bpf program

eBPF maps are created when a program is loaded. In this definition the map is an per-cpu array, but there are a variety of types.

```
struct bpf_map_def SEC("maps") rxcnt = {
        .type = BPF_MAP_TYPE_PERCPU_ARRAY,
        .key_size = sizeof(u32),
        .value_size = sizeof(long),
        .max_entries = 256,
};
```

While executing eBPF program rxcnt map can be accessed like this:

```
long *value;
u32 ipproto = 17;

value = bpf_map_lookup_elem(&rxcnt, &ipproto);
if (value)
        *value += 1;  /* We saw a UDP frame! */
        /* BPF_MAP_TYPE_PERCPU_ARRAY maps does not need to sync between CPUs
         * if using BPF_MAP_TYPE_ARRAY use __sync_fetch_and_add(value, 1); */
```

Find eBPF map file-descriptor in user space

Since eBPF maps can be used to communicate information (statistics in this example) between the eBPF program easily. First locate the map:

```
struct bpf_map *map = bpf_object__find_map_by_name(obj, "rxcnt");
if (!map) {
        printf("finding a map in obj file failed\n");
        return EXIT_FAILURE;
}
map_fd = bpf_map__fd(map);
```

Map file descriptor (map_fd) needed to interactive with BPF-syscall

Reading eBPF map data from user space

Now the map contents can be accessed via map_fd like this:

```
unsigned int nr_cpus = bpf_num_possible_cpus();
__u64 values[nr_cpus];
__u32 key = 17;
__u64 sum = 0;
int cpu;

if (bpf_map_lookup_elem(map_fd, &key, &value))
        return EXIT_FAILURE;

/* Kernel return memcpy version of counters stored per CPU */
for (cpu = 0; cpu < nr_cpus; cpu++)
        sum += values[cpu];

printf("key %u value %llu\n", key, sum);
```

Userspace would sum counters per CPU This allows eBPF kernel program to run faster since not using atomic ops

Advanced XDP Concepts

XDP redirect is powerful

XDP_REDIRECT action is special

XDP action code XDP_REDIRECT is flexible

- In basic form: Redirecting RAW frames out another net_device/ifindex
  - Egress device driver needs to implement ndo_xdp_xmit
- Redirect into map gives flexibility to invent new destinations
  - And allow to hide bulking in bpf map code

Remember use helper: bpf_redirect_map to activate bulking

- Using helper: bpf_redirect_map gives you better performance than bpf_redirect

Inventing redirect types via maps

The devmap: BPF_MAP_TYPE_DEVMAP

- Contains net_devices, userspace adds them via ifindex to map-index

The cpumap: BPF_MAP_TYPE_CPUMAP

- Allow redirecting RAW xdp_frame's to remote CPU
  - SKB is created on remote CPU, and normal network stack invoked
- The map-index is the CPU number (the value is queue size)

AF_XDP - "xskmap": BPF_MAP_TYPE_XSKMAP

- Allow redirecting RAW xdp frames into userspace
  - via new Address Family socket type: AF_XDP

For NIC driver developer

Deep dive into the code behind XDP

- and driver level requirements

Driver XDP RX-handler (called by napi_poll)

Extending a driver with XDP support:

```
while (desc_in_rx_ring && budget_left--) {
        action = bpf_prog_run_xdp(xdp_prog, xdp_buff);
        /* helper bpf_redirect_map have set map (and index) via this_cpu_ptr */
        switch (action) {
         case XDP_PASS:           break;
         case XDP_TX:             res = driver_local_xmit_xdp_ring(adapter, xdp_buff); break;
         case XDP_REDIRECT:       res = xdp_do_redirect(netdev, xdp_buff, xdp_prog);   break;
                                  /*via xdp_do_redirect_map() pickup map info from helper */
         default:                 bpf_warn_invalid_xdp_action(action);              /* fallthrough */
         case XDP_ABORTED:        trace_xdp_exception(netdev, xdp_prog, action);  /* fallthrough */
         case XDP_DROP:           res = DRV_XDP_CONSUMED; break;
        } /* left out acting on res */
}
/* End of napi_poll call do: */
xdp_do_flush_map(); /* Bulk size chosen by map, can store xdp_frame's for flushing */
driver_local_XDP_TX_flush();
```

Bulk via: helper bpf_redirect_map + xdp_do_redirect + xdp_do_flush_map

Restrictions on driver memory model

XDP put certain restrictions on RX memory model

- The one page per RX-frame: No longer true
- Requirement: RX-frame memory must be in continues in physical memory
  - Needed to support eBPF Direct-Access to memory validation
- (Currently) Also require tail-room for SKB shared-info section
  - for SKB alloc outside driver, fits well with driver using build_skb() API

Not supported: drivers that split frame into several memory areas

- This usually result in disabling Jumbo-Frame, when loading XDP prog
- XDP have forced driver to support several RX-memory models
  - This was part of the (evil?) master-plan...

New pluggable memory models per RX queue

Recent change: Memory return API

- API for how XDP_REDIRECT frames are freed or “returned”
  - XDP frames are returned to originating RX driver
- Furthermore: this happens per RX-queue level (extended xdp_rxq_info)

This allows driver to implement different memory models per RX-queue

- E.g. needed for AF_XDP zero-copy mode

Also opportunity to share common RX-allocator code between drivers

- page_pool is an example, need more drivers using it

End

Thanks to all contributors

- XDP + eBPF combined effort of many people