Introduction to: XDP and BPF building blocks

Jesper Dangaard Brouer
Kernel Developer
Red Hat

ebplane hosted by Juniper
USA, Sunnyvale, Oct 2019

The 'ebplane' project

The ebplane project: early startup phase

- Initial presentation title: "Universal Data Plane Proposal"
  - Shows interest in leveraging eBPF technology for networking
- Yet to be defined: use-cases and network-layers to target

This presentation: eBPF technology level setting

- Building blocks and their limitations
- Designing with eBPF requires slightly different thinking...
  - essential for success of this project

Different design thinking: High level overview

Wrong approach: Designing new data plane from scratch

Key insight #1: Modify behaviour of existing system (Linux kernel)

- Via injecting code snippets at different hooks
- BPF code snippets are event-based and by default stateless
- Obtain state and change runtime behaviour via shared BPF-maps

Key insight #2: Only load code when actually needed

- The fastest code is code that doesn't run (or even gets loaded)
- Design system to only load code relevant to user configured use-case
- E.g. don't implement generic parser to handle every known protocol
  - instead create parser specific to user's need/config

Basic introduction and understanding of eBPF

Technical: Getting up to speed on eBPF technology

Basic introduction and understanding of BPF

- eBPF bytecode
- Compiling restricted-C to eBPF
  - compiler stores this in ELF-format
  - which can be loaded into the Linux kernel

eBPF bytecode and kernel hooks

The eBPF bytecode is:

- Generic Instruction Set Architecture (ISA) with C calling convention
  - Read: the eBPF assembly language
- Designed to run in the Linux kernel
  - It is not a kernel module
  - It is a sandbox technology; BPF verfier ensures code safety
  - Kernel provides eBPF runtime environment, via BPF helper calls

Different Linux kernel hooks run eBPF bytecode, event triggered

- Two hooks of special interest: XDP and TC-BPF
- Many more eBPF hooks (tracepoints, all function calls via kprobe)

Compiling restricted-C to eBPF into ELF

LLVM compiler has an eBPF backend (to avoid writing eBPF assembly by hand)

- Write Restricted C – some limits imposed by sandbox BPF-verifier

Compiler produces a standard ELF “executable” file

- Cannot execute this file directly, as the eBPF runtime is inside the kernel
- Need an ELF loader that can:
  - Extract the eBPF bytecode and eBPF maps
  - Do ELF relocation of eBPF map references in bytecode
  - Create/load eBPF maps and bytecode into kernel
- Attaching to hook is a separate step

Recommend using libbpf

Recommend using libbpf as the ELF loader for eBPF

- libbpf is part of Linux kernel tree
- Facebook fortunately exports this to https://github.com/libbpf
  - XDP-tutorial git repo, uses libbpf as git-submodule

Please userspace apps: Everybody should use this library

- Unfortunately several loaders exists
- Worst case is iproute2 has its own
  - causes incompatible ELF object, if using eBPF maps
  - (plan to converting iproute2 to use libbpf)

eBPF concepts: context, maps and helpers

Each eBPF runtime event hook gets a pointer to a context struct

- BPF bytecode has access to context (read/write limited)
  - verifier may adjust the bytecode for safety

The BPF program itself is stateless

- eBPF maps can be used to create state and “config”
- Maps are basically key = value construct

BPF helpers are used for

- Calling kernel functions, to obtain info/state from kernel

Introducing XDP

ebplane: how to leverage eBPF technology for networking

- One option is XDP (eXpress Data Path)
  - When targeting network layers L2-L3
  - L4 use-cases come with some caveats

What is XDP?

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

- New programmable layer in-front of traditional network stack
  - Read, modify, drop, redirect or pass
- For L2-L3 use-cases: seeing x10 performance improvements!
  - Similar speeds as DPDK
- Can accelerate in-kernel L2-L3 use-cases (e.g. forwarding)

What is AF_XDP? (the Address Family XDP socket)

- Hybrid kernel-bypass facility via XDP_REDIRECT filter
- Delivers raw L2 frames into userspace (in SPSC queue)

What makes XDP different and better?

Not bypass, but in-kernel fast-path

The killer feature of XDP is integration with Linux kernel,

- Leverages existing kernel infrastructure, eco-system and market position
- Programmable flexibility via eBPF sandboxing (kernel infra)
- Flexible sharing of NIC resources between Linux and XDP
- Kernel maintains NIC drivers, easy to deploy everywhere
- Cooperation with netstack via eBPF-helpers and fallback-handling
- No need to reinject packets (unlike bypass solutions)

AF_XDP for flexible kernel bypass

- Cooperate with use-cases needing fast raw frame access in userspace
- No kernel reinject, instead choose destination before doing XDP_REDIRECT

Simple view on how XDP gains speed

XDP speed gains come from

- Avoiding memory allocations
  - no SKB allocations and no-init (only `memset(0)` of 4 cache-lines)
- Bulk processing of frames
- Very early access to frame (in driver code right after DMA sync)
- Ability to skip (large parts) of kernel code

Skipping code: Efficient optimisation

Encourage adding helpers instead of duplicating data in BPF maps

Skipping code: Imply skipping features provided by network stack

- Gave users freedom to e.g. skip netfilter or route-lookup
- But users have to re-implement features they actually needed
  - Sometimes cumbersome via BPF-maps

To avoid re-implementing features:

- Evolve XDP via BPF helpers that can do lookups in kernel tables
- Example of BPF-helpers available today for XDP:
  - FIB routing lookup
  - Socket lookup

XDP actions and cooperation

What are the basic XDP building blocks you can use?

BPF programs return an action or verdict, for XDP there are 5:

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

Ways to cooperate with network stack

- Pop/push or modify headers: Change RX-handler kernel use
  - e.g. handle protocol unknown to running kernel
- Can propagate 32 bytes of metadata from XDP stage to network stack
  - TC (cls_bpf) hook can use metadata, e.g. set SKB mark
- XDP_REDIRECT map special, can choose where netstack “starts/begin”
  - CPUMAP redirect starts netstack on remote CPU
  - veth redirect starts inside container

XDP redirect

ebplane very likely needs redirect feature

- XDP redirect is an advanced feature
  - Requires some explanation to fully grasp why map variant is novel

Basic: XDP action XDP_REDIRECT

XDP action code XDP_ REDIRECT

- In basic form: Redirecting RAW frames out another net_device via ifindex
- Egress driver: must implement ndo_xdp_xmit (and ndo_xdp_flush)

Need to be combined with BPF-helper calls, two variants

- Different performance (single CPU core numbers, 10G Intel ixgbe)
- Using helper: bpf_redirect = 7.5 Mpps
- Using helper: bpf_redirect_map = 13.0 Mpps

What is going on?

- Using redirect maps gives a HUGE performance boost, why!?

Redirect using BPF-maps is novel

Why is it so brilliant to use BPF-maps for redirecting?

Named “redirect” as more generic, than “forwarding”

- Tried to simplify changes needed in drivers, process per packet

First trick: Hide RX bulking from driver code via BPF-map

- BPF-helper just sets map+index, driver then calls xdp_do_redirect() to read it
- Map stores frame in temporary store (curr bulk per 16 frames)
- End of driver NAPI poll “flush” - calls xdp_do_flush_map()
- Extra performance benefit: from delaying expensive NIC tailptr/doorbell

Second trick: invent new types of redirects easy

- Without changing any driver code! - Hopefully last XDP action code

Redirect map types

Note: Using redirect maps require extra setup step in userspace

The "devmap": BPF_MAP_TYPE_DEVMAP + BPF_MAP_TYPE_DEVMAP_HASH

- Contains net_devices, userspace adds them using ifindex as map value

The "cpumap": BPF_MAP_TYPE_CPUMAP

- Allow redirecting RAW xdp frames to remote CPU - map-index is CPU#
  - SKB is created on remote CPU, and normal network stack "starts"

AF_XDP - "xskmap": BPF_MAP_TYPE_XSKMAP

- Allows redirect of RAW xdp frames into userspace - map-index usually RXq#
  - via new Address Family socket type: AF_XDP

Introducing TC-BPF

ebplane: leverage eBPF technology for networking

- Another option is using TC (Traffic Control) BPF-hooks
  - When targeting network layers L4-L7
  - L2-L3 are of course still possible

What is TC-BPF?

The Linux TC (Traffic Control) layer has some BPF-hook points

- In TC filter 'classify' step: both ingress and egress
- Scalable: runs outside TC-root lock (with preempt disabled + RCU read-side)

Operates on SKB context object (struct __sk_buff)

- Pros: netstack collaboration easier, rich access to SKB features
- Pros: easier L4, and (via sockmap) even L7 filtering in-kernel
- Pros: more BPF-helpers available
- Cons: Slower than XDP due to SKB alloc+init and no-bulking

TC-BPF actions or verdicts

TC-BPF progs are usually used in 'direct-action' (da) mode

- Similar to XDP, BPF-prog will directly return TC-action code (TC_ACT_*)

BPF (cls_bpf) semantic for some of the available TC_ACT_* codes:

- TC_ACT_ OK: pass SKB onwards (and set skb->tc_index)
- TC_ACT_ UNSPEC: multi-prog case, continue to next BPF-prog
- TC_ACT_ SHOT: drop SKB (kfree_skb) and inform caller NET_XMIT_DROP
- TC_ACT_ STOLEN: drop SKB (consume_skb()) inform NET_XMIT_SUCCESS
- TC_ACT_ REDIRECT: redirect packet to another net_device (bpf_redirect())

TC-BPF access to packet-data memory

TC-BPF also (like XDP) has direct access to packet-data (read: fast)

- But access limited to memory-linear part of packet-data
- Thus, how much is accessible depends on how SKB were created
- BPF-helper bpf_skb_pull_data() can be used, but very expensive

XDP also has direct access, but forces drivers to use specific memory model

- Requires packet-data to be delivered "memory-linear" (in physical mem)

Cooperation between XDP and TC-BPF

XDP and TC-BPF can both run and collaborate via:

- Shared BPF maps as state or config
- XDP metadata (in front of packet) available to TC-BPF (as already mentioned)
- TC-BPF can function as fallback layer for XDP

XDP is lacking TX hook

- For now, TC egress BPF hooks have solved these use-cases

Design perspective

Higher level: Design perspective

- from a BPF view point

BPF view on: data-plane and control-plane

This covers both XDP and TC networking hooks

Data-plane: inside kernel, split into:

- Kernel-core: Fabric in charge of moving packets quickly
- In-kernel eBPF program:
  - Policy logic decide action (e.g. pass/drop/redirect)
  - Read/write access to packet

Control-plane: in userspace

- Userspace loads eBPF program
- Can control program via changing BPF maps
- Everything goes through bpf system call

BPF changing the kABI landscape

kABI = Kernel Application Binary Interface

Distros spend a lot of resources maintaining kABI compatibility

- to satisfy out-of-tree kernel modules, calling kernel API / structs
- e.g. tungsten contrail-vrouter kernel module hook into RX-handler (L2)

BPF offers a way out, with some limits due to security/safety:

- Fully programmable hooks points (restricted-C)
- Access sandboxed e.g. via context struct and BPF-helpers available
- Possible policy actions limited by hook

Userspace “control-plane” API tied to userspace app (not kernel API)

In principle: BPF-instruction set and BPF-helpers are still kABI

Designing with BPF for XDP+TC

Examples of designing with BPF

Design protocol parser with BPF for XDP/TC

Background: XDP/TC metadata area placed in-front packet headers (32 Bytes). Works as communication channel between XDP-tail-calls, and into TC-BPF hook

Split BPF-prog parser-step into standalone BPF-prog

- Output is parse-info with header types and offsets
- Parse-info is stored in XDP/TC metadata area (in-front packet headers)

Tail-call next BPF-prog, which has access to metadata area

- Due to verifier, prog getting parse-info still need some bounds checking

Advantage: Parser prog can be replaced by hardware

Design to load less-code

Generic netstack is also slow because

- Needs to handle every known protocol (cannot fit in Instruction-Cache)

BPF gives ability to runtime change and load new code

- The 'ebplane' design should take advantage of this

Specifically for: Protocol parsing "module"

- Don't create huge BPF-prog that can parse everything
- Idea: Domain Specific Language (maybe P4) for BPF-prog parsing step
  - Users describe protocols relevant for them, and parse-info struct
  - Result: smaller BPF-prog for parsing (less Instruction-Cache usage)
  - (Make sure this can also be compiled for HW targets)

Containers

Relating XDP and TC-BPF to Containers

TC-BPF for containers

Containers are in most cases better handled via TC-BPF

- Have to allocate SKB anyway for delivery in container

Advanced use-case are possible with TC-BPF, like

- Allows for L4-L7 policy enforcement for microservices
- See CloudFlare blogpost: via sockmap + strparser
- Kernel level proxy service, also handling TLS/HTTPS via ktls+sockmap

The Cilium project has already demonstrated this is doable

- Even fully integrated with Kubernetes CNI

XDP for containers

In general: XDP redirect into container doesn't make sense

- veth driver support redirect, but will just create SKB later
  - why not just take the SKB alloc overhead up-front?

XDP-redirect into veth, only makes sense if re-redirecting

- E.g. veth might not be final destination
- Could call this service chaining containers

Imagine: Packaging L2-L3 appliances as containers

- Like Suricata for inline Intrusion Prevention (IPS)
- Virtual IP-router or firewall appliance

Pitfalls and gaps

Explicitly covering known gaps

- when leveraging eBPF technology for networking

Gaps: **IP-fragmentation not handled**

Issue: (L3) IP-fragments doesn't contain (L4) port numbers (e.g. TCP/UDP)

- Challenge for UDP tunnels and L4 load-balancers

Neither XDP nor TC-BPF do IP-defrag

- IP-defrag happens later at Transport Layer (L4)

As TC-BPF works with SKBs, would be possible to

- Extend with BPF-helper to do the IP-defrag
  - Not enough demand to get this implemented

In practice: People configure MTU to avoid IP-fragmentation

Alternative: Fallback to network stack to handle IP-defrag

Gaps: **XDP broadcast and multicast**

Cloning packets in XDP is not currently possible

XDP: Sending to multiple destination; not supported

- Simple idea: Broadcast via redirect “send” to ALL port in devmap
- Multicast via creating a devmap per multicast group

Alternative is to fallback

- Let either: netstack or TC-BPF hook handle broadcast/multicast

Gaps: XDP doesn't handle multi-buffer packets

This limits XDP max packet size

- XDP max MTU: 3520 bytes (page_size(4096) - headroom(256) - shinfo(320))

Multi-frame packets have several use cases

- Jumbo-frames (larger than 3520 bytes)
- TSO (TCP Segmentation Offload)
- Header split, (L4) headers in first segment, (L7) payload in next

XDP proposal for multi-frame packets

- Design idea/proposal in XDP-project: xdp-multi-buffer01-design.org

Gaps: Getting XDP driver features

BPF core reject loading BPF-prog using features kernel don't have

- Features can be probed via cmdline: bpftool feature probe

XDP features also depends on driver code

- If native-XDP can load, usually XDP_DROP + ABORTED + PASS works
- XDP_REDIRECT not supported by all drivers, cannot detect this
  - At runtime packets dropped and kernel-log contains WARN_ONCE

Work in-progress covered at LPC2019 talk: XDP: the Distro View

Topic: Testing

How to test XDP and BPF programs?

Testing available in kernel tree

Kernel have BPF code examples in directory samples/bpf

- Red Hat QA do use these samples as testing tools for XDP

Kernel also have selftests in tools/testing/selftests/bpf

- These kind test might be better for 'ebplane' project?

How to test XDP

After veth got native-XDP support (v4.20), easiest test is

- Writing scripts that setup veth namespaces for testing
- Example: kernel/tools/testing/selftests/bpf/test_xdp_vlan.sh

Alternative use bpf_prog_test_run() (bpf-syscall BPF_PROG_TEST_RUN)

- Instead of attach: “run” loaded BPF-prog with bpf_prog_test_run()
  - Provide constructed packet data input + output buffer
- Example: kernel/tools/testing/selftests/bpf/prog_tests/xdp.c
  - Hint: pkt_v4 and pkt_v6 defined in tools/testing/selftests/bpf/test_progs.c

XDP community

Status on XDP and BPF communities

State of XDP community

XDP developer community:

- Part of Linux kernel, under both netdev and bpf subsystems
- BPF developers also coordinate under IOvisor (LF-project)
- XDP-project keeps track of work items

XDP users: Community and resources

- Mailing list for newbies: xdp-newbies@vger.kernel.org
- Getting started with XDP-tutorial: full build and testlab environment
  - Simply git clone and run make: https://github.com/xdp-project/xdp-tutorial
- Cilium maintains official: BPF and XDP Reference Guide

End: Summary

Kernel now have eBPF programmable network fast-path

- that can now compete with kernel-bypass speeds

FOSS community needs projects like 'ebplane'

- to build projects and products with this technology

Hopefully this presentation gave enough information

- in form of building blocks and known limitations

XDP-project coordination:

- https://github.com/xdp-project/xdp-project

Bonus slides

Topic: XDP redirect into Guest-VM

Makes sense: Using XDP for Guest-VM redirect

- Allow skipping overhead of Host-OS network stack