BPF and XDP Explained

Nic Viljoen & Simon Horman
DXDD
Utrecht, 8th June 2017
Objectives

Give user a basic understanding of the architecture of eBPF
- What is it
- The programming model
- The kernel hooks

Give user a basic understanding of XDP
- What is it/Where is it
- How to use it (beginner level!)
- How to offload it
What is eBPF?

eBPF is a simple way to extend the functionality of the kernel at runtime

- Effectively a small kernel based machine
  - 10 64bit registers
  - 512 byte stack
- Data structures known as maps (unlimited size)
- 4K BPF instructions (Bytecode)
- Verifier to ensure kernel safe
  - no loops, not more than 4K insns, not more than 64 maps etc...
- Can be JITed to ensure maximum performance
Those who have publically stated they are using BPF or are planning to use BPF include

- Facebook-Load Balancing, Security
- Netflix-Network Monitoring
- Cilium Project
- Cloudflare-Security
- OVS-Virtual Switching

Due to its upstream safety and kernel support BPF provides a safe, flexible and scalable networking tool
The Programming Model

LLVM is used to compile from supported languages
- C
- Go
- P4

When Programs are loaded
- Verifier is called-ensure safety
- Program is JITed-ensure perf
- Can also be offloaded
  - nfp_bpf_jit upstream
Maps-What They Are

Maps are key value stores
- Can be accessed from kernel or user space
- Used for interaction between kernel and user space programs

Number of different types of maps
- Used for interaction between kernel and user space programs

```c
enum bpf_map_type {
    BPF_MAP_TYPE_UNSPEC,
    BPF_MAP_TYPE_HASH,
    BPF_MAP_TYPE_ARRAY,
    BPF_MAP_TYPE_PROG_ARRAY,
    BPF_MAP_TYPE_PERF_EVENT_ARRAY,
    BPF_MAP_TYPE_PERCPU_HASH,
    BPF_MAP_TYPE_PERCPU_ARRAY,
    BPF_MAP_TYPE_STACK_TRACE,
    BPF_MAP_TYPE_CGROUP_ARRAY,
    BPF_MAP_TYPE_LRU_HASH,
    BPF_MAP_TYPE_LRU_PERCPU_HASH,
};
```
Creating Maps

- Option 1: create map with syscall
  - `bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr))`
- Option 2: define a struct `bpf_map_def` with an elf section
  - `__attribute__ SEC("maps")`-also uses syscall!

```c
int bpf_create_map(enum bpf_map_type map_type,
                   unsigned int key_size,
                   unsigned int value_size,
                   unsigned int max_entries)
{
    union bpf_attr attr = {
        .map_type   = map_type,
        .key_size   = key_size,
        .value_size = value_size,
        .max_entries= max_entries
    };

    return bpf(BPF_MAP_CREATE, &attr, sizeof(attr));
}
```

```c
struct bpf_map_def SEC("maps") my_map = {
    .type       = BPF_MAP_TYPE_XXX,
    .key_size   = sizeof(u32),
    .value_size = sizeof(u64),
    .max_entries= 42,
    .map_flags  = 0
};
```
eBPF Bytecode: Quick Overview

```
eBPF Bytecode: op:8, dst_reg:4, src_reg:4, off:16, imm:32
```

- op code is divided into the sections
  - Operation code (4bits) e.g BPF_MOV, BPF_JNE
  - Source bit (1 bit) BPF_X (use src_reg and dst_reg) or BPF_K (use dst_reg and 32 bit imm)
  - instruction class (3 bits) e.g BPF_ALU, BPF_ALU64, BPF_JMP

```
BPF_MOV | BPF_X | BPF_ALU64, 0x6, 0x1, 0x0000, 0x00000000
```
- Move contents of register 1 to register 6

```
BPF_JNE | BPF_K | BPF_JMP, 0x1, 0x0, 0x0011, 0x00008100
```
- Jump 11 insns forward-can also jump backwards-if contents of register 1 is not equal to 0x00008100
BPF Kernel Hooks

Many hooks with different purposes

- kprobes
- socket filters-tcpdump-old school!
- seccomp
- netfilter (new)
- TC
- XDP(no skb-super fast!)

XDP will be our focus for the rest of this talk
XDP

BPF hook in the driver
- Allows for high speed processing before skb is attached to packet
- Currently 4 return codes: XDP_ABORT, XDP_DROP, XDP_PASS, XDP_TX
- XDP_REDIRECT in the pipeline
- Usecases include DDoS protection and load balancing
- Includes maximum of 256 bytes of prepend
- Metadata is just pointers to start of packet and end

```c
struct xdp_md {
    __u32 data;
    __u32 data_end;
};
```
Program Example (xdp1_kern.c)

Simple drop example

- Note the use of standard header infrastructure
- Associated user space program maintaining a set of counters
- I am not going to go through line by line-for more detail check out Andy and Jesper’s awesome tutorial-in links
- Will come back to this example later on…

This can be found in the recent (4.8+) kernels at linux/samples/bpf
Optimizing XDP

A simple checklist—not comprehensive!

▪ Ensure BPF JIT is enabled
▪ Pin queues to interfaces
▪ Set ringsize to an optimal level for your NIC and application
▪ To gain some idea of your NIC’s driver based XDP performance check simple XDP_DROP and XDP_TX programs
▪ Many people use single core performance as a reasonable benchmark
  ▪ To do this use the ethtool -X command
  ▪ You will NOT get the simple program performance if you build something complex (Duh)
Offloading XDP

Netronome have upstreamed the initial version of the nfp_bpf_jit

▪ More to come!
  ▪ Maps
  ▪ Compiler optimizations
  ▪ Magic
Offload Architecture

- **BPF syscall**
  - user space
  - kernel space
  - program
  - type (sk filter, kprobe, cls, xdp)
  - license
  - ...

- **BPF prog**
  - fd

- **TC**
  - cls_bpf

- **XDP ctrl**
  - fd, skip_* flags

- **tc**
  - fd, skip_* flags

- **host JIT**
  - offload object

- **verification**
  - modification

- **HW JIT / translator**
  - setup tc
  - ndo
  - stats & maps

- **RX**
  - TX

- **driver**
References

Kernel Docs: https://www.kernel.org/doc/Documentation/networking/filter.txt
Initial XDP Presentation: https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf
Andy and Jesper’s Talk: https://netdevconf.org/2.1/slides/apr7/gospodarek-Netdev2.1-XDP-for-the-Rest-of-Us_Final.pdf
Reading List: https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/
Search: google.com :)

Thanks!

ANY QUESTIONS?