Introduction to eBPF
(for network packet processing)

Quentin Monnet
<quentin.monnet@6wind.com>
@qeole
BPF example with tcpdump

```c
# tcpdump -i eth0 tcp dst port 22 -d

(000) ldh [12]    # Ethertype
(001) jq #0x86dd    jt 2    jf 6    # is IPv6?
(002) ldb [20]     # IPv6 next header field
(003) jq #0x6       jt 4    jf 15   # is TCP?
(004) ldh [56]     # TCP dst port
(005) jq #0x16      jt 14   jf 15   # is port 22?
(006) jq #0x800     # IPv4?
(007) ldb [23]     # IPv4 protocol field
(008) jq #0x6       jt 9    jf 15   # is TCP?
(009) ldh [20]     # IPv4 flags + frag. offset
(010) jset #0x1fff   jt 15   jf 11   # fragment offset is != 0?
(011) ldxb 4*([14]&0xf)  # x := 4 * header_length (words)
(012) ldh [x + 16]  # TCP dest port
(013) jq #0x16      jt 14   jf 15   # is port 22?
(014) ret #262144  # trim to 262144 bytes, return packet
(015) ret #0       # drop packet
```

tcpdump → libpcap → BPF bytecode → kernel interpreter / JIT

**BPF filter** attached to socket to filter packets and avoid useless copies
History

- 1993: “cBPF” (*classic* BPF) on BSD, for packet filtering
- 1997: ported to Linux
BPF ~ Basics

BPF is an assembly-like language with registers and stack, integer arithmetic, conditional branches. JIT- compilable, for performances.

Usage: filter packets **in the kernel** with programs coming **from user space**

```c
int s = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);
setsockopt(s, SOL_SOCKET, SO_ATTACH_FILTER, &bpf_prog, sizeof(bpf_prog));
```

Safety ensured by in-kernel verifier:

- No backward jumps
- Program limited to 4096 instructions
- Dynamic packet-boundary checks
- Etc.
Re-designing BPF: extended BPF

History

- 1993: “cBPF” (classic BPF) on BSD, for packet filtering
- 1997: ported to Linux
- 2013+: “eBPF” (extended BPF), Linux only — Project IO Visor

Design goals: better safety, flexibility and performances
How does eBPF improve over cBPF?

- **Technical upgrades**
  - From 2 registers (32-bit) to 11 registers (64-bit)
  - New, larger set of instructions, closer to assembly
  - Etc.

- **New functionalities**
  - **Call instruction**: can call certain (white-listed) kernel helper functions
  - **Tail calls**, kind of “long jumps” into another eBPF program
  - Can **map memory** to communicate with userland applications or other eBPF programs
New hooks... Lots of them!
How to use eBPF?

- **C source code**: `bpf_prog.c`
- **ELF-compiled BPF**: `bpf_prog.o`
- **User program (tc / bcc tools...)**

- **User program**
- **LLVM/clang**
- **Verifier**
- **Maps**
  - Array
  - Hashmap
  - LPM
  - ...

- **BPF program attached and run**

*bcc tools*: C helpers + Python wrappers to help handling BPF programs
*Also*: Go, Lua helpers; P4 to eBPF-compatible C compiler; ...

**Userspace**

**Kernel**
Example, for tc (traffic control) interface:

```c
/* Drop all packets for TCP port 22 */
#define BLOCKED_TCP_PORT 22

int handle_ingress(struct __sk_buff *skb) {

    /* Variable declaration & initialization omitted here */
    ...

    /* Length check */
    if (data + sizeof(*eth) + sizeof(*iph) + sizeof(*tcp) > data_end)
        return TC_ACT_OK; /* Forward */

    /* Is it IPv4? */
    if (eth->h_proto != htons(ETH_P_IP))
        return TC_ACT_OK; /* Forward */

    /* Is it TCP? Is IP header length equal to 5? */
    if (iph->protocol != IPPROTO_TCP || iph->ihl != 5)
        return TC_ACT_OK; /* Forward */

    /* Is it the port we want to block? */
    if (tcp->dest == htons(BLOCKED_TCP_PORT))
        return TC_ACT_SHOT; /* Drop */

    return TC_ACT_OK; /* Forward */
}
```
Compile and run

Compile from C to eBPF:

```
$ clang -O2 -emit-llvm -c bpf_prog.c -o - | \ 
  llc -march=bpf -filetype=obj -o bpf_prog.o
```

Attach it as a tc classifier

```
# tc qdisc add dev eth0 clsact
# tc filter add dev eth0 ingress \ 
  bpf direct-action object-file bpf_prog.o
```

If needed, initialize the maps (user-space program with bpf() syscall)
eXpress DataPath (XDP): in-kernel fast packet processing:

- Hooks in supported drivers to attach eBPF programs
- Intercepts packets before packet reaches the stack / before sk_buff allocation
- For basic use cases. Complex use cases: forward to the stack

Linux 4.8+; Still in development
XDP

Packets

Net device

Sockets

Network stack

tc ingress

tc egress

Net device

Userspace

Kernel

Quentin Monnet (6WIND) | Introduction to eBPF
XDP

Userspace

Kernel

DPDK

Packets

Network stack

Sockets

Net device

tc ingress

tc egress

Net device
XDP performances

XDP benchmark, single CPU:

- Filter drop all (but read/touch data): 20 Mpps
- TX-bounce forward: 12 Mpps
- TX-bounce with UDP + MAC rewrite: 10 Mpps

CPU @3.70 GHz; Mellanox 40 Gbps, mlx4 driver, with DDIO
XDP_DDoS_protecting_osd2017.pdf

Hardware offload exists
Use cases for eBPF/XDP ~ Some network functions

- Protection against DDoS attacks
- Load balancing
- QoS
- ILA (Identifier-Locator Addressing) router
- ...
Use cases for eBPF/XDP ~ Virtual switch

- A new backend for Open vSwitch
- BEBA project: fast and stateful packet processing for vSwitches

Quentin Monnet (6WIND) | Introduction to eBPF
Use cases for eBPF/XDP ~ Network fabric

- Orchestration
  - Control layer: Policy, ACLs, BPF program generation, ...
  - Userspace
  - Kernel Maps
  - BPF programs: routing steps
- Fabric setup
- States

- Open Virtual Network (OVN) backend with IO Modules
- Cilium: Fast networking for containers with BPF/XDP
Summary

- eBPF is fast, stateful
- Runs in kernel, with userspace flexibility
- XDP: in-kernel dataplane acceleration
- Networking, but also Linux tracing / monitoring
- Still under development, growing community
Questions
Resources

GitHub repository of the **IO Visor** project (bcc tools, documentation, and more)
https://github.com/iovisor/

Resources on BPF — *Dive into BPF: a list of reading material*
https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

BEBA research project
http://www.beba-project.eu/

Cilium (code repository, links to presentations), initially scheduled on this slot
https://github.com/cilium/cilium