Traffic filtering at scale on Linux

François Serman <*francois.serman@corp.ovh.com*>

Pass The Salt 2018

- Introduction
- (past) BPF
- (present) eBPF
- Let's play with BPF!
- Performance analysis
- Summary and conclusion

Introduction

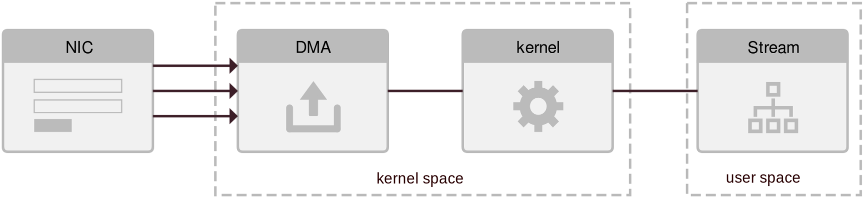
whoami

```
fserman@ovh $ groups
dev vac
fserman@ovh $ uptime | awk '{ print $2, $3, $4 }'
up 435 days,

fser@home $ groups
clx, lautre.net, hexpresso
```

Back to the presentations

- Traffic filtering:
  - Obviously: classify packets we want to keep, drop the rest ;
  - Achieved using (e)BPF.
- at scale:
  - Tenth of gigabits per seconds ;
  - Millions of packets per seconds ;
  - We'll see how to generate such traffic ;
  - but also how to mitigate it (XDP).
- on Linux:
  - Using recent (> 4.8) kernel facilities.

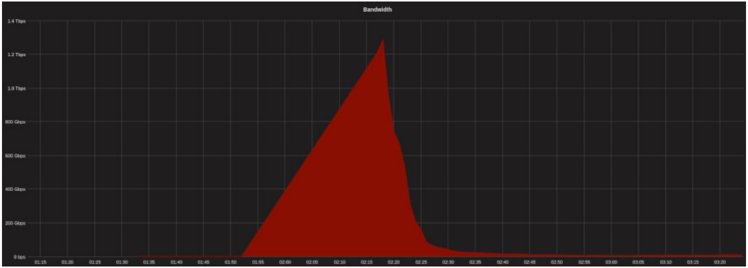
Networking 101



Packets are copied aggregated and ordered towards userspace

Breadcrumb

Top amplification attack on Memcached (UDP 11211) : 1.3Tbps.
(For the record: MIRAI was 1Tbps)



The amplification attack aiming Memcached in march 2018.

(past) BPF

199[23] : Steven McCanne & Van Jacobson at Berkeley

The BSD Packet Filter: A New Architecture for User-level Packet Capture

Steven McCanne & Van Jacobson – Lawrence Berkeley Laboratory[1]

ABSTRACT

Many versions of Unix provide facilities for user-level packet capture, making possible the use of general purpose workstations for network monitoring. Because network monitors run as user-level processes, packets must be copied across the kernel/user-space protection boundary. This copying can be minimized by deploying a kernel agent called a *packet filter*, which discards unwanted packets as early as possible. The original Unix packet filter was designed around a stack-based filter evaluator that performs sub-optimally on current RISC CPUs. The BSD Packet Filter (BPF) uses a new, register-based filter evaluator that is up to 20 times faster than the original design. BPF also uses a straightforward buffering strategy that makes its overall performance up to 100 times faster than Sun's NIT running on the same hardware.

Provide a way to filter packets and avoid useless packets copies (kernel to user).

Main concepts

- [Efficient] Kernel architecture for packet capture;
  - Discard unwanted packets as early as possible;
  - Packet data references should be minimised;
  - Decoding an instruction $\sim$ single C switch statement;
  - Abstract machine registers should reside in physical one;
- Protocol independent: no modification to the kernel to support a new protocol;
- General: instruction set should be rich enough to handle unforeseen uses;

BPF is a virtual machine

What is a virtual machine?

- Abstract computing machine;
- Has its own instruction-set, registers, memory representation;
- Cannot run directly on actual hardware:
- Hence need a VM loader and interpreter or compiler.

The BPF virtual machine

All values are 32 bits (instructions / data)
Fixed-length instructions:

- **Load** data to registers;
- **Store** data to memory;
- **ALU instructions** arithmetic or logic operations;
- **Branch instructions** alter the control-flow based on a test;
- **Return instructions** terminate the filter;
- **(Misc operations)**

Usage

Most famous use case:

- **tcpdump** (via **libpcap**).
- cls_bpf (TC classifier for shaping)
- xt_bpf (iptables module).

Please tcpdump, show us all **UDP** packets towards **memcached**.

```
# tcpdump -p -d 'ip and udp and dst port 11211'
```

Notice the difference with/without *«ip and»*

Under the hood

```
# tcpdump -p -d 'ip and udp and dst port 11211'
(000) ldh      [12]
(001) jeq      #0x800          jt 2    jf 10
(002) ldb      [23]
(003) jeq      #0x11           jt 4    jf 10
(004) ldh      [20]
(005) jset     #0x1fff         jt 10   jf 6
(006) ldxb     4*([14]&0xf)
(007) ldh      [x + 16]
(008) jeq      #0x2bcb         jt 9    jf 10
(009) ret      #262144
(010) ret      #0
```

Decrypting the output

- `(000) ldh        [12]`
  Load half-word from packet at offset 12 (EtherType)

Decrypting the output

- `(000) ldh [12]`
  Load half-word from packet at offset 12 (EtherType)
- `(001) jeq #0x800 jt 2 jf 10`
  If equals 0x800 (EtherType IPv4). If true, go to 2, else to 10.

OVH

Decrypting the output

- `(000) ldh          [12]`
  Load half-word from packet at offset 12 (EtherType)
- `(001) jeq          #0x800                jt 2    jf 10`
  If equals 0x800 (EtherType IPv4). If true, go to 2, else to 10.
- `(002) ldb          [23]`
  Load double-word at offset 23 (Protocol field in IPv4 header)

OVH

Decrypting the output

- `(000) ldh [12]`
  Load half-word from packet at offset 12 (EtherType)
- `(001) jeq #0x800 jt 2 jf 10`
  If equals 0x800 (EtherType IPv4). If true, go to 2, else to 10.
- `(002) ldb [23]`
  Load double-word at offset 23 (Protocol field in IPv4 header)
- `(003) jeq #0x11 jt 4 jf 10`
  If proto is UDP, continue to 4, else go to 10

Decrypting the output

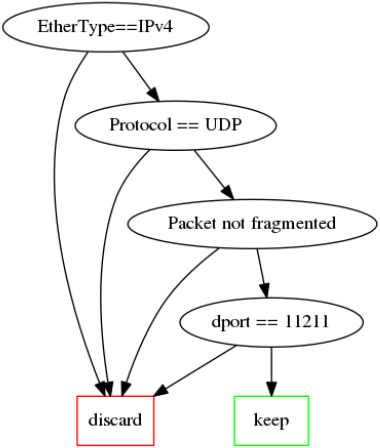
- `(000) ldh [12]`
  Load half-word from packet at offset 12 (EtherType)
- `(001) jeq #0x800 jt 2 jf 10`
  If equals 0x800 (EtherType IPv4). If true, go to 2, else to 10.
- `(002) ldb [23]`
  Load double-word at offset 23 (Protocol field in IPv4 header)
- `(003) jeq #0x11 jt 4 jf 10`
  If proto is UDP, continue to 4, else go to 10
- `(007) ldh [x + 16]`
  Load UDP Dest port

Decrypting the output

- `(000) ldh [12]`
  Load half-word from packet at offset 12 (EtherType)
- `(001) jeq #0x800 jt 2 jf 10`
  If equals 0x800 (EtherType IPv4). If true, go to 2, else to 10.
- `(002) ldb [23]`
  Load double-word at offset 23 (Protocol field in IPv4 header)
- `(003) jeq #0x11 jt 4 jf 10`
  If proto is UDP, continue to 4, else go to 10
- `(007) ldh [x + 16]`
  Load UDP Dest port
- `(008) jeq #0x2bcb jt 9 jf 10`
  If dest port == 11211 (0x2bcb), go to 9, else go to 10

Visualization

```
tcpdump -p -d 'ip and udp and dst port 11211'
```



(present) eBPF

Improvements ($\sim$ 2013)

From Documentation/networking/filter.txt:

- Registers:
  - Increase number of registers from 2 to 10;
  - 64 bits formats;
  - ABI mapped on the underlying architecture;
- Operations in 64 bits;
- Conditionnal jt/jf replaced with jt/fall-through;
- BPF calls;
- Maps

eBPF today

- the old BPF is refered to as classic BPF (cBPF);
- eBPF is the new BPF!
- No longer limited to packet filtering:
  - tracing (kprobes);
  - security (seccomp);
  - ...

eBPF today

- BPF is very suitable for *JIT* (Just In Time compilation):
  - Virtual registers already map the physicals one;
  - Only have to issue the proper instruction;
  - Available for x86_64, arm64, ppc64, s390x, mips64, sparc64 and arm;
  - 1 C switch statement became 1 instruction.
- BPF bytecode is **verified** before loading in the kernel.
- Hardened JIT available.

```
# echo 1 > /proc/sys/net/core/bpf_jit_enable
```

eBPF verifier

Provides a verdict whether the bytecode is safe to run:

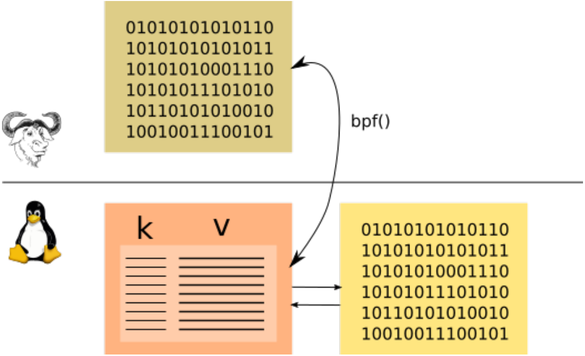
- a BPF program must **always** terminate:
  - size-bounded (max 4096 instr);
  - Loop detections (CFG validation);
- a BPF program must be safe:
  - detecting out of range jumps
  - detecting out of bonds r/w
  - context-aware: verifying helper function call's arguments
  - ...

Refere to *kernel/bpf/verifier.c*.

OVH

eBPF Maps (1/3)

Generic storage facility for sharing data between kernel and userspace.



Interract via *bpf()* syscall (lookup/update/delete).
Helpers available on *tools/lib/bpf/bpf.h*.

eBPF Maps (2/3)

Defined by:

- types (as of 4.18 19 types):
  - **Arrays** *BPF_MAP_TYPE_ARRAY* (+ PERCPU);
  - **Hashes** *BPF_MAP_TYPE_HASH* (+PERCPU);
  - **LRU** *BPF_MAP_TYPE_LRU_HASH* (+PERCPU);
  - **LPM** *BPF_MAP_TYPE_LPM_TRIE*;
- max number of elements
- key size in bytes
- value size in bytes

Let's play with BPF!

In kernel tools

Have a look on *samples/bpf*:

- bpf_asm a minimal cBPF assembler;
- bpf_dbg a small debugger for cBPF programs;
- bpftool a generic tool to interract with eBPF programs:
  - show dump load pin programs
  - show create pin update delete maps
  - ...

BPF Compiler Collection (BCC)

Quoting their README:

- "Toolkit for creating efficient kernel tracking and manipulation programs [...]"
- "it makes use of extended BPF".

For us:

- Provides a way to load BPF code (not only for networking)
- Collection of BPF programs (traces, perf...)
- Python API

Demo time

Collect statistics on running memcached.

- One party generates memcached requests (randomly);
- The other party has two parts:
  - kernel part: parses the protocol, extracts the request's keyword, and updates counters;
  - userspace part: periodicaly displays the counters.

Memcached commands:

add append cas decr delete flush_all get gets incr prepend replace stats

```
$ wc -l *
   30 flood.py
  188 xdp_memcached.c
  144 xdp_memcached.py
```

Performance analysis

Some numbers

- Achieving high bandwidth is “easy”
- Handling lots of packets is harder:
  - For 64bytes pkts ($\sim$ 80 on the wire)
    - 10Gbps : 14.8Mpps
    - 25Gbps : 37.0Mpps
    - 50Gbps : 74.0Mpps
    - 100Gbps: 148.0Mpps
  - For 1500 bytes pkts:
    - 10Gbps : 820Kpps
    - 25Gbps : $\sim$ 2Mpps
    - 50Gbps : $\sim$ 4.1Mpps
    - 100Gbps: $\sim$ 8.2Mpps

Experimental setup

- Two servers : one sender and one receiver
  - 2 * Intel(R) Xeon(R) Gold 6134 CPU @ 3.20GHz (8c/16t)
  - 12 * 8Gb (= 96Gb) DDR4
  - Mellanox MT27700 (50Gbps ConnectX-4)
  - Linux v4.15
- back to back (no switch was harmed for this presentation)

Objectives:

- Increase PPS
- Reduce packet loss

Produce modern graphs

Install the following packages:

- InfluxDB
- Telegraf
- Grafana

Import dashboard **928**.
Done.

State of the art Yolo devops

```
# wget https://dl.influxdata.com/influxdb/releases/ \
  influxdb_1.1.1_amd64.deb
# wget https://dl.influxdata.com/telegraf/releases/ \
  telegraf_1.1.2_amd64.deb
# wget https://s3-us-west-2.amazonaws.com/ \
  grafana-releases/release/grafana_5.1.4_amd64.deb

# dpkg -i *.deb

# sed -i 's/^# \(\[\[inputs\.net\]\]\)/\1/' \
  /etc/telegraf/telegraf.conf

# systemctl start {influxdb,telegraf,grafana-server}.service
```

Generating traffic

We'll cover several methods to generate traffic. You'll have to guess the rate (in pps) for each:

- `while true; do nc ... ; done`
- `python flood.py`
- scapy
- tcpreplay
- C threaded program
- kernel's pktgen
- DPDK's pktgen

netcat (code)

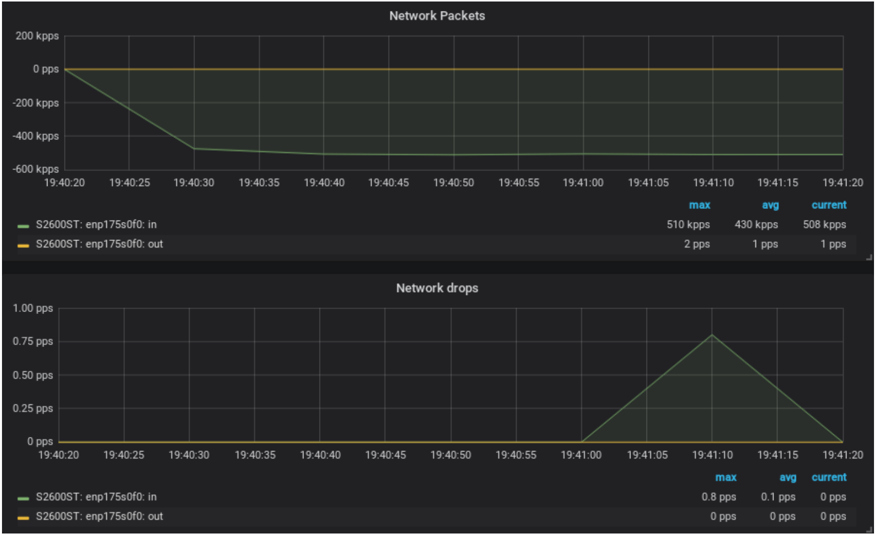
```
while true ; do
  ( echo 'Hello, world!' |
    nc -w 1 -u 10.0.1.2 $((RANDOM %65534)) & )
done
```

netcat (outcome)

python (code)

```
import socket

UDP_IP, UDP_PORT = "10.0.1.2", 5005
MESSAGE = "Hello, World!"

if len(sys.argv) == 2:
    UDP_PORT = int(sys.argv[1])

sock = socket.socket(socket.AF_INET, # Internet
                     socket.SOCK_DGRAM) # UDP
while True:
   sock.sendto(MESSAGE, (UDP_IP, UDP_PORT))
```

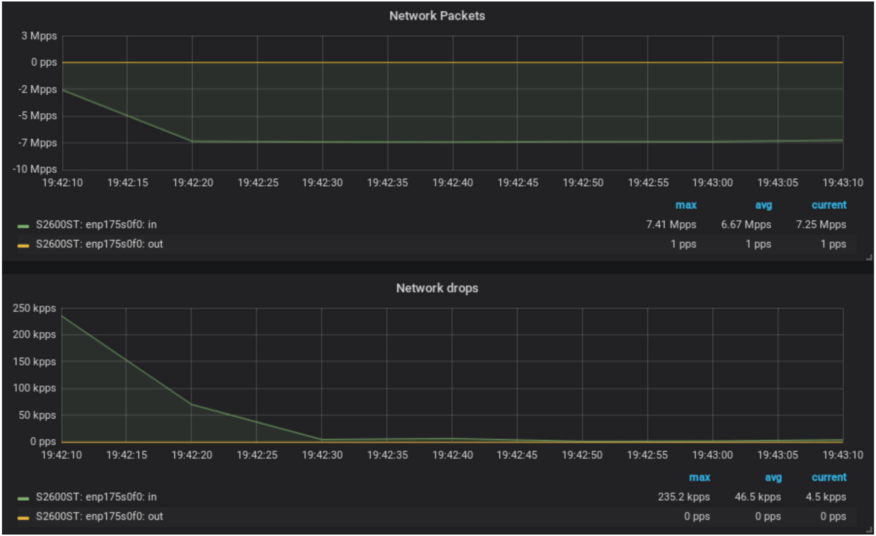
python (outcome)



python (multiple processes)

```
for i in {4000..4032} ; do
  ( python flood.py ${i} & )
done
```

python multiple processes (outcome)



scapy (code)

```
send(IP(dst="10.0.1.2")/UDP(dport=123), loop=100000)
```

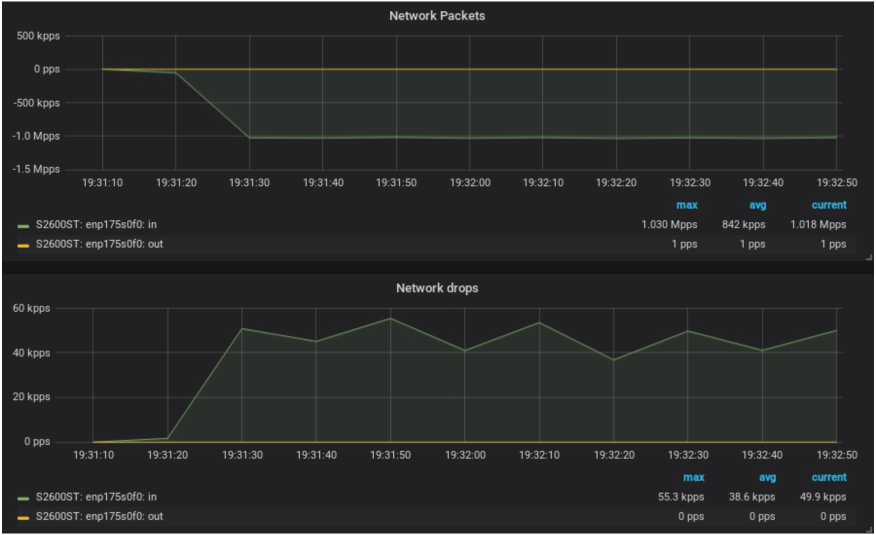
scapy (outcome)

tcpreplay (code)

```
>>> wrpcap("/tmp/batch.pcap",
           Ether(dst="7c:fe:90:57:ab:c8")
           / IP(src="10.0.1.1",dst="10.0.1.2")
           / UDP(dport=123) * 1000)
# tcpreplay -i enp134s0f0 --loop 5000000 -tK /tmp/batch.pcap
```

Where -*t* stands for “topspeed” and k ...

tcpreplay (outcome)



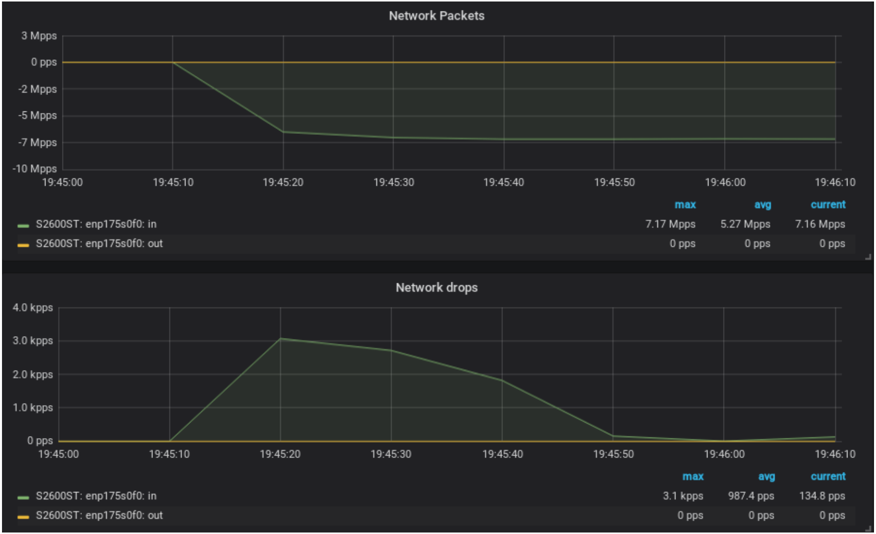
C threaded program (code)

- https://github.com/vbooter/DDoS-Scripts/blob/master/UDP.c
- (minor modification)

```
# ./UDP 10.0.1.2 4242 0 64 32
```

- 0 is the throttle
- 64 the packet size
- 32 the number of threads

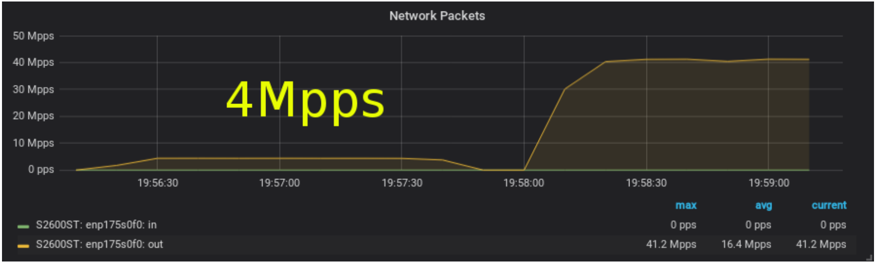
C threaded program (outcome)



kernel's pktgen (config)

```
# cd ~/linux/sample/pktgen
# export PGDEV=/proc/net/pktgen/enp175s0f0@0

# ./pktgen_sample05_flow_per_thread.sh -i enp175s0f0 \
  -s 64 -d 10.0.1.1 -m 7c:fe:90:57:ab:c0 -n 0

and

./pktgen_sample05_flow_per_thread.sh -i enp175s0f0 \
  -s 64 -d 10.0.1.1 -m 7c:fe:90:57:ab:c0 -n 0 -t 32
```

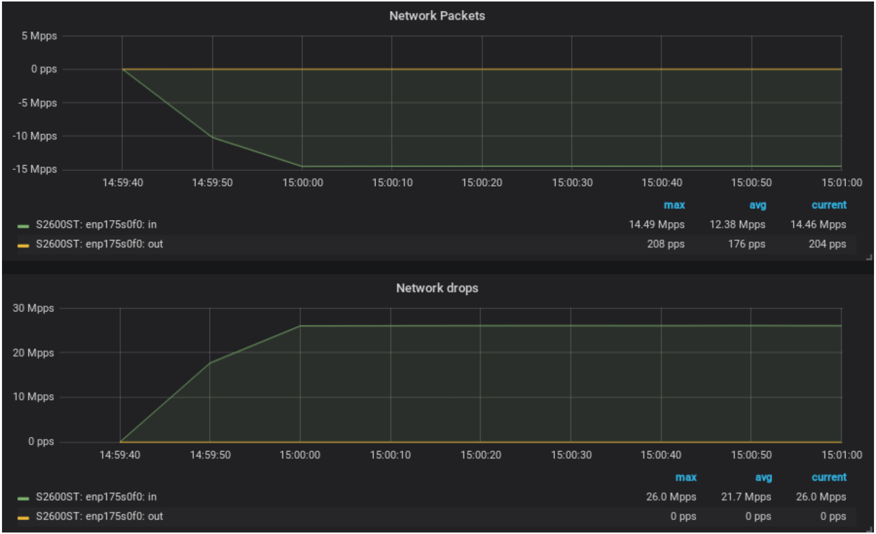
kernel's pktgen (outcome)



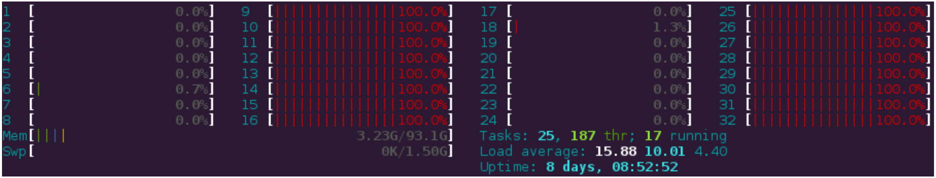
DPDK's pktgen (config)

```
enable 0 range
range 0 dst ip 10.0.1.2 10.0.1.2 10.0.1.254 0.0.0.1
range 0 src ip 10.0.1.3 10.0.1.3 10.0.1.254 0.0.0.1
range 0 proto udp
range 0 dst port 1 1 65534 1
range 0 src port 1 1 65534 1
range 0 dst mac 7c:fe:90:57:ab:c8 7c:fe:90:57:ab:c8
                7c:fe:90:57:ab:c8 00:00:00:00:00:00
```

DPDK's pktgen (outcome)



How does the receiver feel?

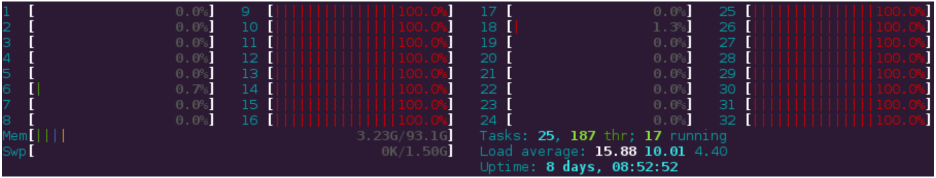


With iptables

```
# iptables -A INPUT -p udp -m udp -j DROP
```

With iptables

```
# iptables -A INPUT -p udp -m udp -j DROP
```



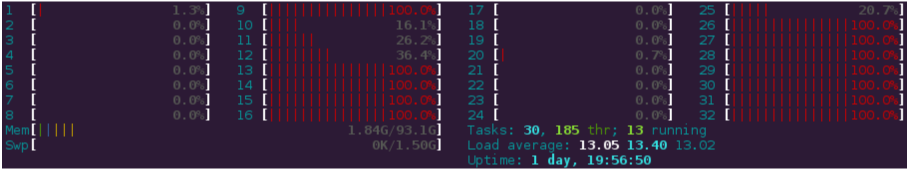
Can we do better?

Can we do better?

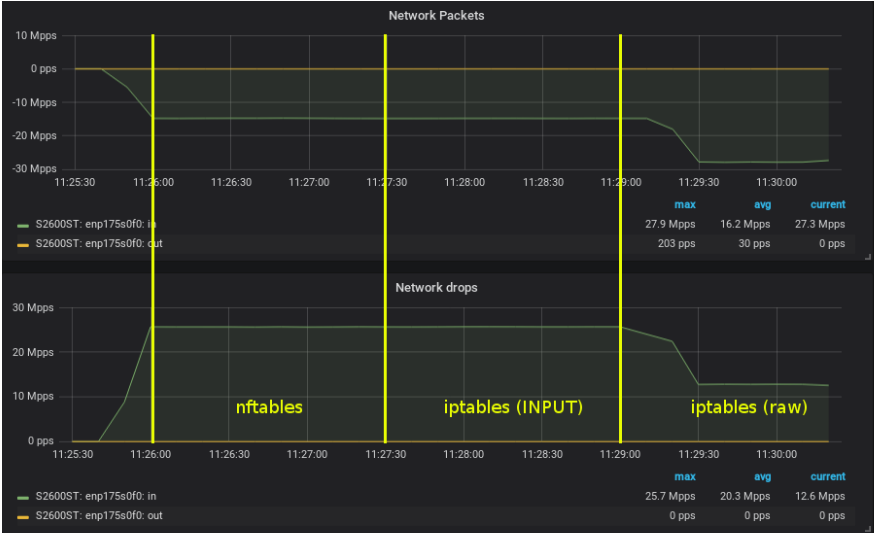
```
# iptables -t raw -A PREROUTING -p udp -m udp -j DROP
```

Can we do better?

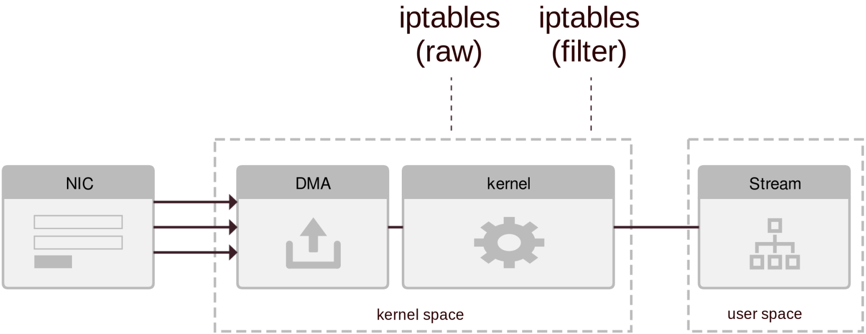
```
# iptables -t raw -A PREROUTING -p udp -m udp -j DROP
```



nftables and iptables



synthesis

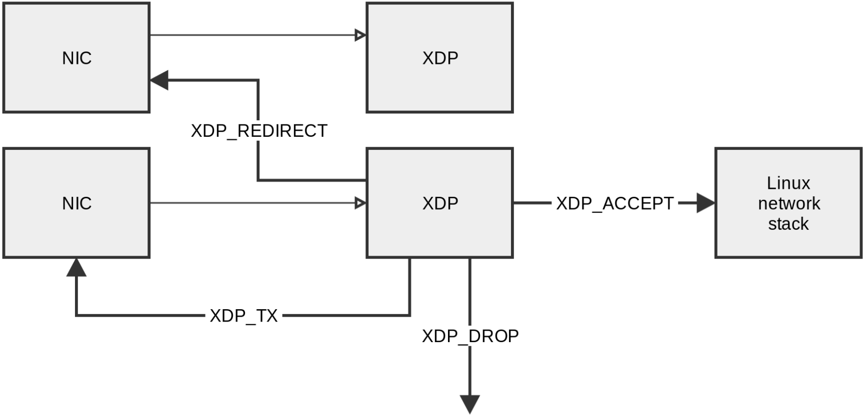

iptables
(raw)
iptables
(filter)
NIC
DMA
kernel
Stream
kernel space
user space

Not the expected result

«Iptables is not slow. It's just executed too late in the stack.»
– (r) Gilberto Bertin

Introduce XDP : What is XDP?

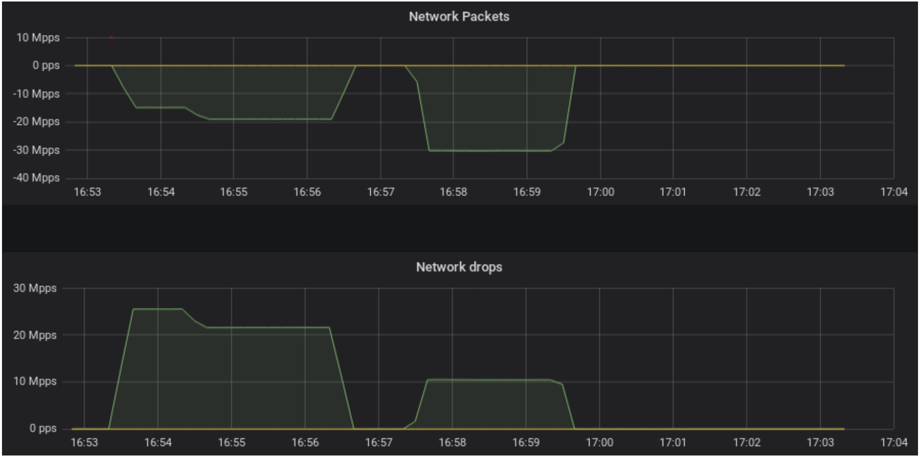
- XDP stands for eXpress Data Path.
- Programmable, High-performances, specialized application, packet processor in the linux networking stack.



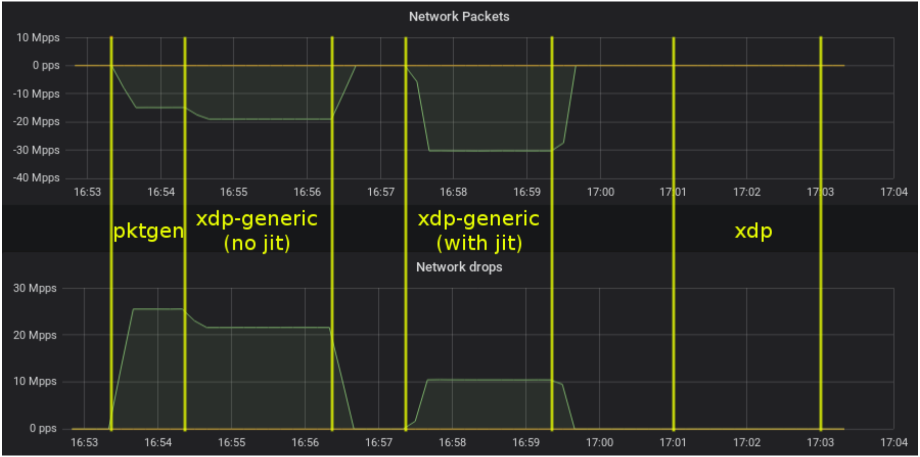
XDP : eXpress Data Path

- XDP is *not*:
  - a replacement for TCP/IP stack
  - kernel bypass
- Runs eBPF program on hooks:
  - In the kernel (TC/xdp-generic)
  - In driver (xdp or xdpoffload) => before **skb** allocation
- 3 outcomes:
  - Accept the packet: XDP_PASS
  - Drop the packet: XDP_DROP
  - Redirect the packet: XDP_TX or XDP_REDIRECT

XDP


Network Packets
10 Mpps
0 pps
-10 Mpps
-20 Mpps
-30 Mpps
-40 Mpps
16:53
16:54
16:55
16:56
16:57
16:58
16:59
17:00
17:01
17:02
17:03
17:04
Network drops
30 Mpps
20 Mpps
10 Mpps
0 pps
16:53
16:54
16:55
16:56
16:57
16:58
16:59
17:00
17:01
17:02
17:03
17:04

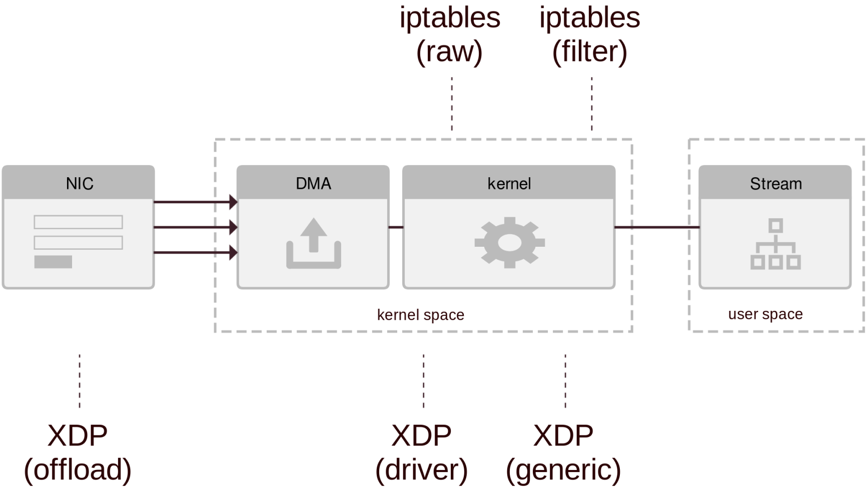
XDP


Network Packets
10 Mpps
0 pps
-10 Mpps
-20 Mpps
-30 Mpps
-40 Mpps
16:53
16:54
16:55
16:56
16:57
16:58
16:59
17:00
17:01
17:02
17:03
17:04
pktgen
xdp-generic
(no jit)
xdp-generic
(with jit)
xdp
Network drops
30 Mpps
20 Mpps
10 Mpps
0 pps

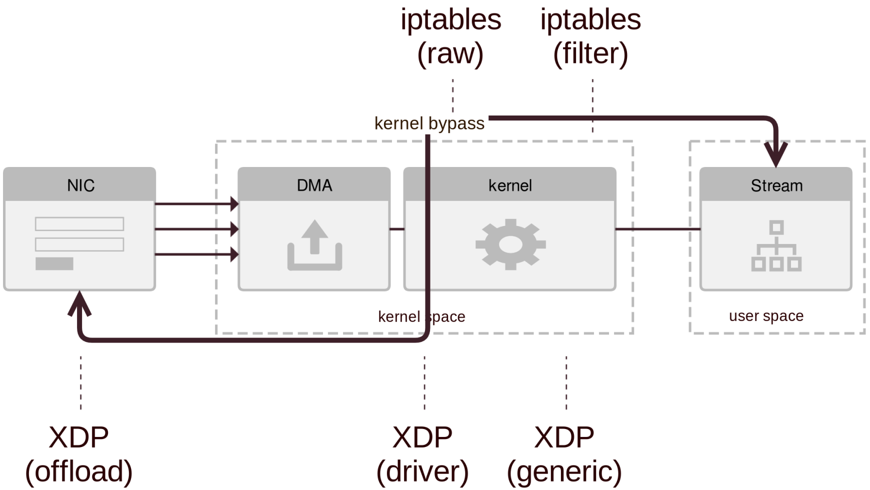
Minimal example

```
#include <linux/bpf.h>

#ifndef __section
# define __section(NAME)                  \
   __attribute__((section(NAME), used))
#endif

__section("prog")
int xdp_drop(struct xdp_md *ctx)
{
    return XDP_DROP;
}

char __license[] __section("license") = "GPL";
```

Synthesis


iptables
(raw)
iptables
(filter)
NIC
DMA
kernel
Stream
kernel space
user space
XDP
(offload)
XDP
(driver)
XDP
(generic)

XDP alternatives: kernel bypass



Kernel bypass

- PF_RING
- NetMap
- DPDK
- ...
- Pros:
  - Fast!
- Cons:
  - Require driver support
  - Handle the whole stack “by hand”
  - NIC may be dedicated (not visible from the Linux).

Summary and conclusion

What we have seen

- Scaling traffic is not trivial;
- Filters need to be applied as early as possible;
- XDP is a standard (as in mainline integrated) way;
- But alternatives exist.

Issues with XDP

- Require “recent” software stack
  - kernel
  - iproute
  - toolchain (LLVM for instance)
- Complex
  - Basically have to know C
- Increasing number of tools
  - bpfilter
  - bcc
  - P4

Try it yourself

Fork me on github : https://github.com/fser/pts-2018

References

- https://jvns.ca/blog/2017/04/07/xdp-bpf-tutorial/
- https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/
- https://cilium.readthedocs.io/en/latest/bpf/
- https://www.iovisor.org/technology/xdp
- http://prototype-kernel.readthedocs.io/en/latest/bpf/index.html
- man pages:
  - tc-bpf (8)
  - man bpf (2)
- Documentation/networking/filter.txt
- Several netdev-conference's slides.

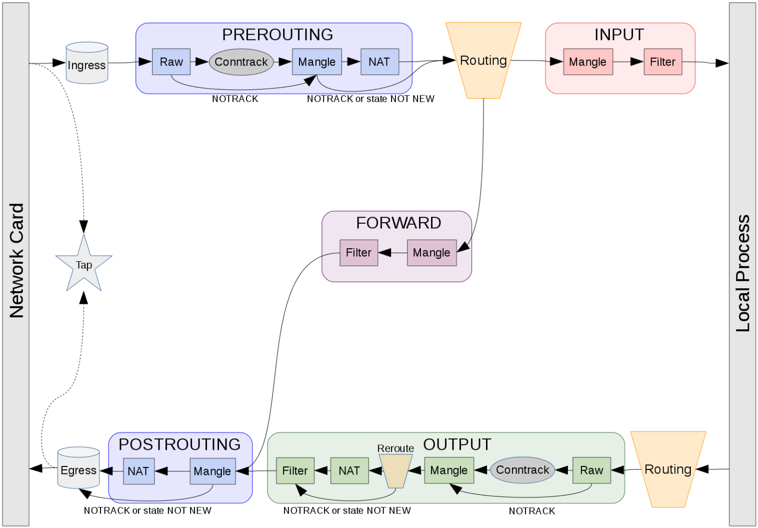
Questions

Backup slides

Loading an XDP program

```
# ip link set dev DEVICE xdp \
           obj OBJECT_FILE.o [ sec SECTION_NAME ]

# tc qdisc add dev DEVICE clsact
# tc filter add dev DEVICE ingress bpf da obj OBJECT_FILE.o
```

Iptables overview



Flood memcached commands

```
#!/usr/bin/env python

import sys, socket, random

UDP_IP, UDP_PORT = "127.0.0.1", 11211
MESSAGE = "\x00\x00\x00\x00\x00\x01\x00\x00{}\r\n"

cmds = '''add append cas decr delete flush_all
get gets incr prepend replace stats'''.split()

sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
while True:
  cmd = random.choice(cmds)
  sock.sendto(MESSAGE.format(cmd), (UDP_IP, UDP_PORT))
```


OVH

XDP parsing - bcc

```
#!/usr/bin/env python

from bcc import BPF

...

b = BPF(src_file="xdp_memcached.c", cflags=["-w",
    "-DRETURNCODE=%s" % ret, "-DCTXTYPE=%s" % ctxtype])

b.attach_xdp(device, fn, flags)

dropcnt = b.get_table("dropcnt")
```

Licenses

Memcached traffic viewer: Apache License, Version 2.0
XDP UDP drop: GPL v2
Scripts & ansible: WTFPL
Slides