Fast Packet Processing using eBPF and XDP

Prof. Marcos A. M. Vieira
mmvieira@dcc.ufmg.br
DCC - UFMG

EVComp 2020

Who is already using eBPF?

June 2018, Layer 4 Load Balancing at Facebook
Katran
https://github.com/facebookincubator/katran/

February 2018, BPF comes through firewalls
https://lwn.net/Articles/747551/
https://lwn.net/Articles/747504/
https://www.netronome.com/blog/frnog-30-faster-networking-la-francaise/

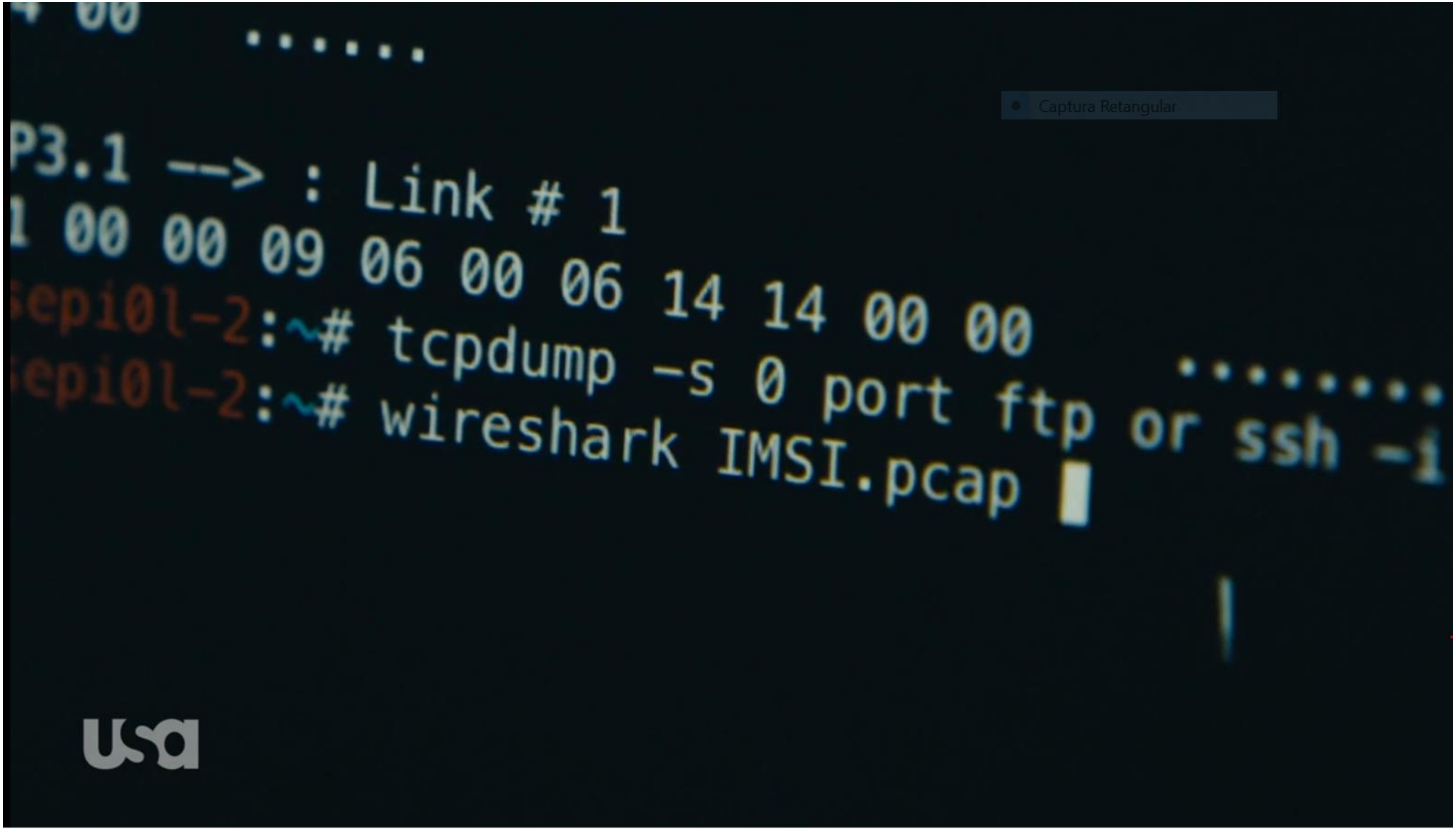
March 2018, Introducing AF_XDP support (to bring packets from NIC driver directly to userspace)
https://lwn.net/Articles/750293/
http://mails.dpdk.org/archives/dev/2018-March/092164.html
https://twitter.com/DPDKProject/status/1004020084308836357

April 2018, Add examples of ipv4 and ipv6 forwarding in XDP (to exploit the Linux routing table to forward packets in eBPF)
https://patchwork.ozlabs.org/patch/904674/

Tcpdump

- Packet analyzer
- Original use-case: tcpdump filter for raw packet sockets
- Libpcap: captures packets
- Might apply BPF-filter

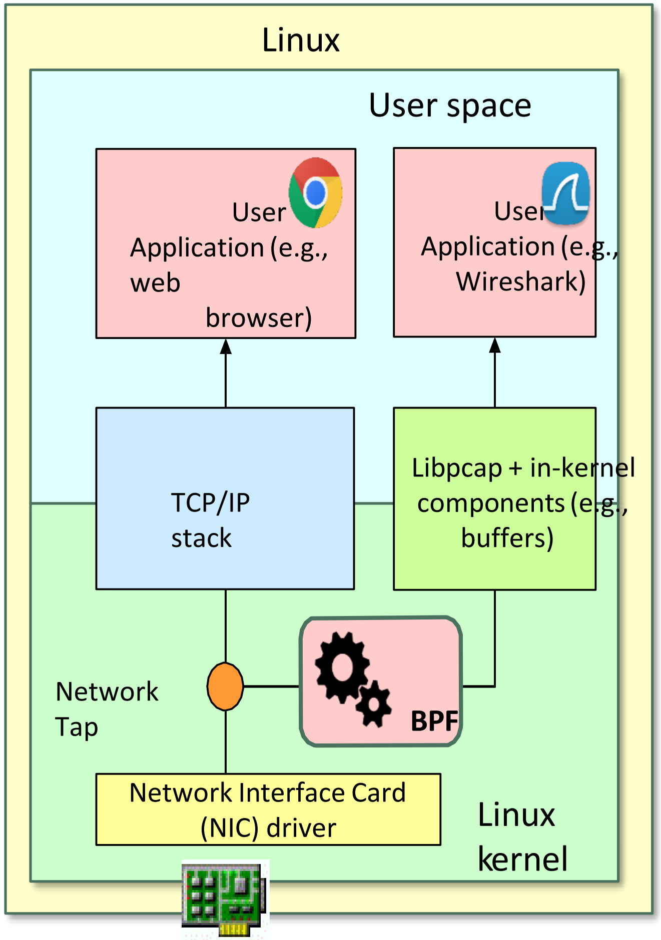
MR. ROBOT - tcpdump



```
4 00
      ......

P3.1 --> : Link # 1
1 00 00 09 06 00 06 14 14 00 00
                                                    ........
sepi0l-2:~# tcpdump -s 0 port ftp or ssh -i
sepi0l-2:~# wireshark IMSI.pcap
```

Berkeley Packet Filter (BPF)

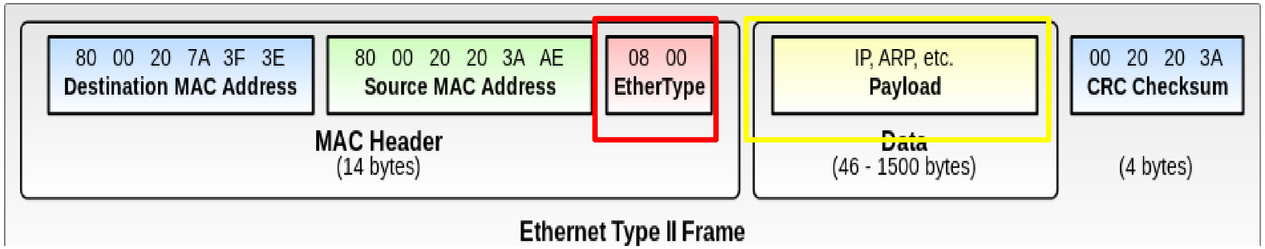
- Generic **in-kernel**, **event-based** **virtual CPU**
  - Introduced in Linux kernel 2.1.75 (1997)
  - Initially used as packet filter by packet capture tool tcpdump (via libpcap)
- In-kernel
  - No syscalls overhead, kernel/user context switching
- Event-based
  - Network packets
- Virtual CPU


Linux
User space
User Application (e.g., web browser)
User Application (e.g., Wireshark)
TCP/IP stack
Libpcap + in-kernel components (e.g., buffers)
Network Tap
BPF
Network Interface Card (NIC) driver
Linux kernel

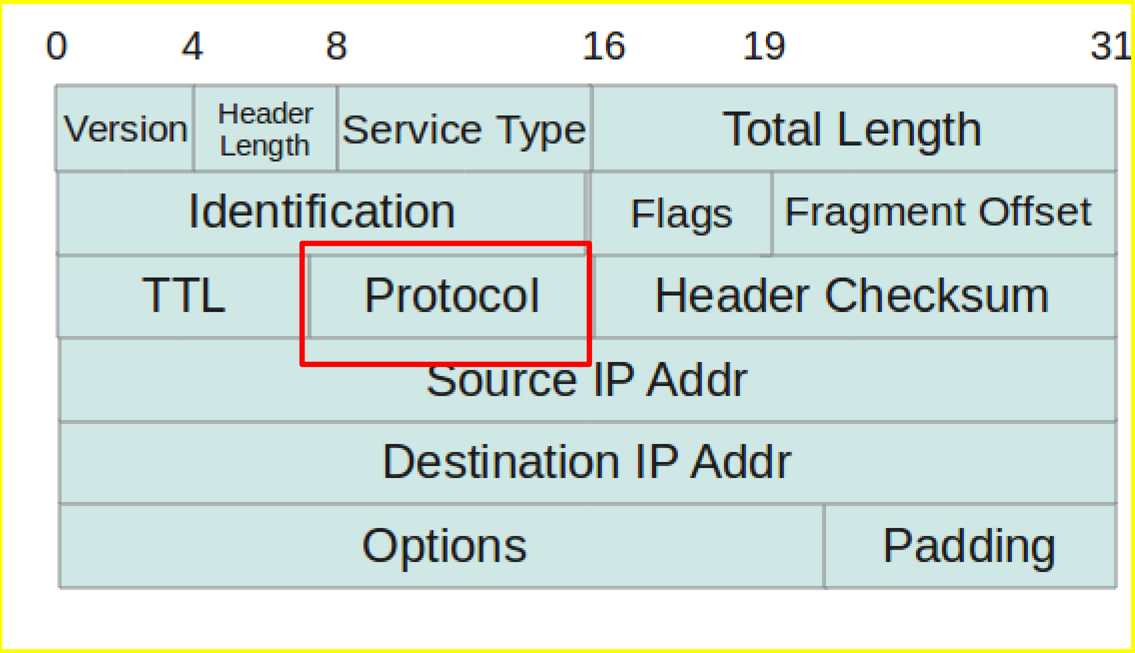
What is Berkeley Packet Filter (BPF)?

- tcpdump -i eno1 –d IPv4_TCP_packet

```
ldh [12]
jne #0x800, drop
ldb [23]
jneq #6, drop
ret #-1
drop: ret #0
```

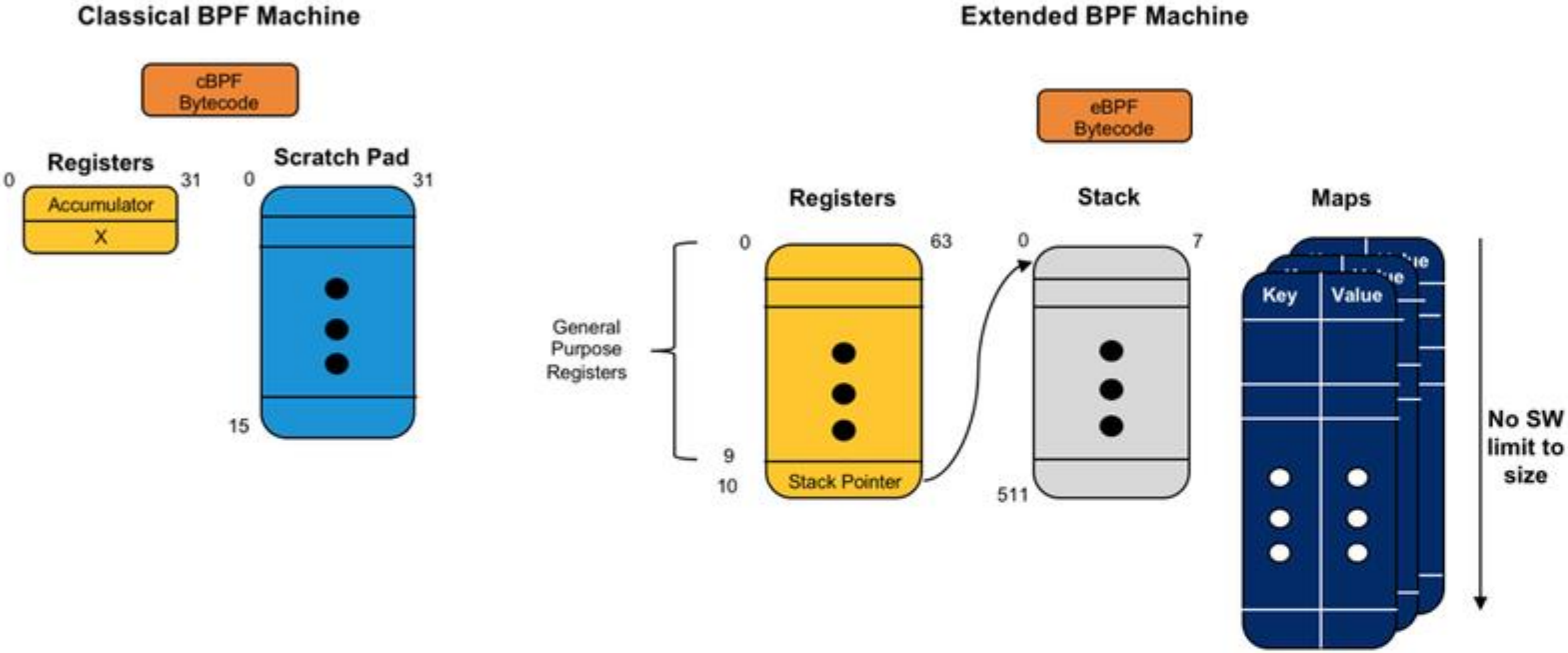


80 00 20 7A 3F 3E
Destination MAC Address
80 00 20 20 3A AE
Source MAC Address
08 00
EtherType
MAC Header
(14 bytes)
IP, ARP, etc.
Payload
Data
(46 - 1500 bytes)
00 20 20 3A
CRC Checksum
(4 bytes)
Ethernet Type II Frame



0 4 8 16 19 31
Version
Header Length
Service Type
Total Length
Identification
Flags
Fragment Offset
TTL
Protocol
Header Checksum
Source IP Addr
Destination IP Addr
Options
Padding

BPF vs. eBPF machines



- Number of registers increase from 2 to 11
- Register width increases from 32-bit to 64-bit
- Conditional jt/jf targets replaced with jt/fall-through

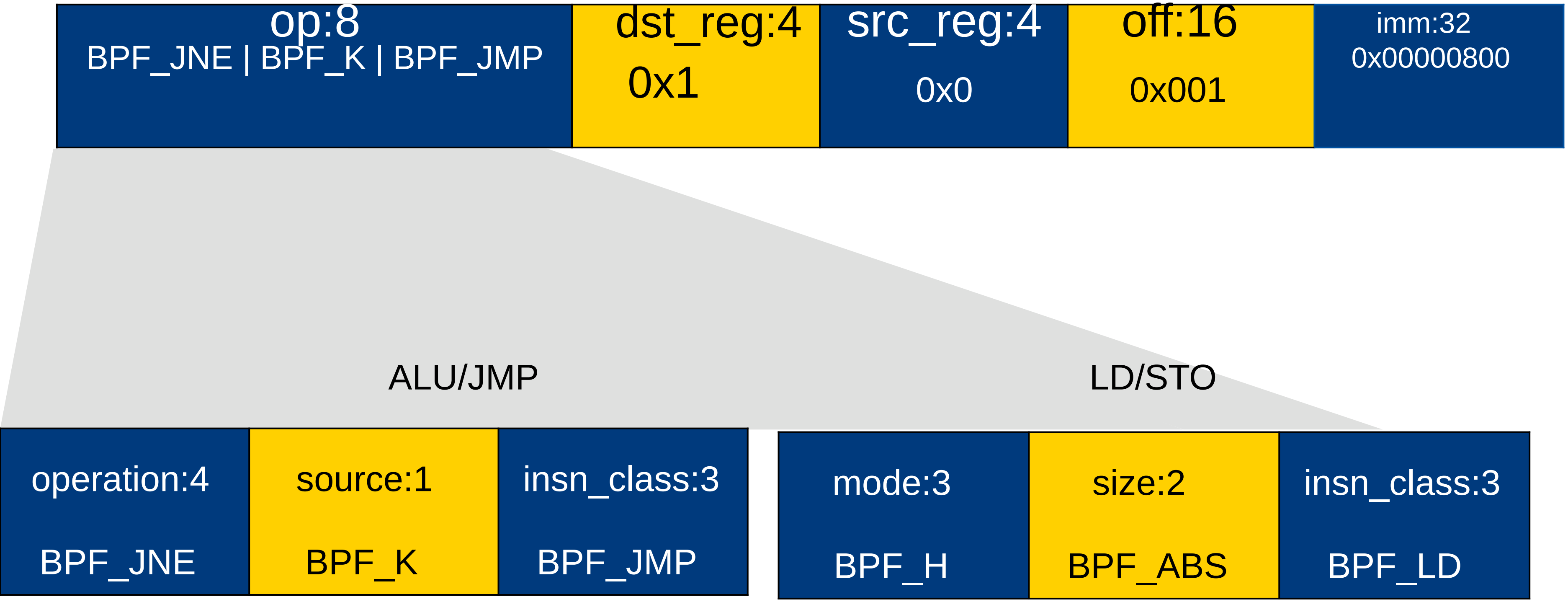
- 11 64-bit registers, 512 bytes stack
- Instructions 64-bit wide

eBPF Instruction Set

- 7 classes:
- BPF_LD, BPF_LDX: hold instructions for byte / half-word / word / double-word load operations.
- BPF_ST, BPF_STX: Both classes are for store operations.
- BPF_ALU: ALU operations in 32 bit mode
- BPF_ALU64: ALU operations in 64 bit mode.
- BPF_JMP: This class is dedicated to jump operations. Jumps can be unconditional and conditional.

eBPF Bytecode

64-bit, 2 operand BPF bytecode instructions are split as follows



op:8	dst_reg:4	src_reg:4	off:16	imm:32
BPF_JNE | BPF_K | BPF_JMP	0x1	0x0	0x001	0x00000800

ALU/JMP

operation:4	source:1	insn_class:3
BPF_JNE	BPF_K	BPF_JMP

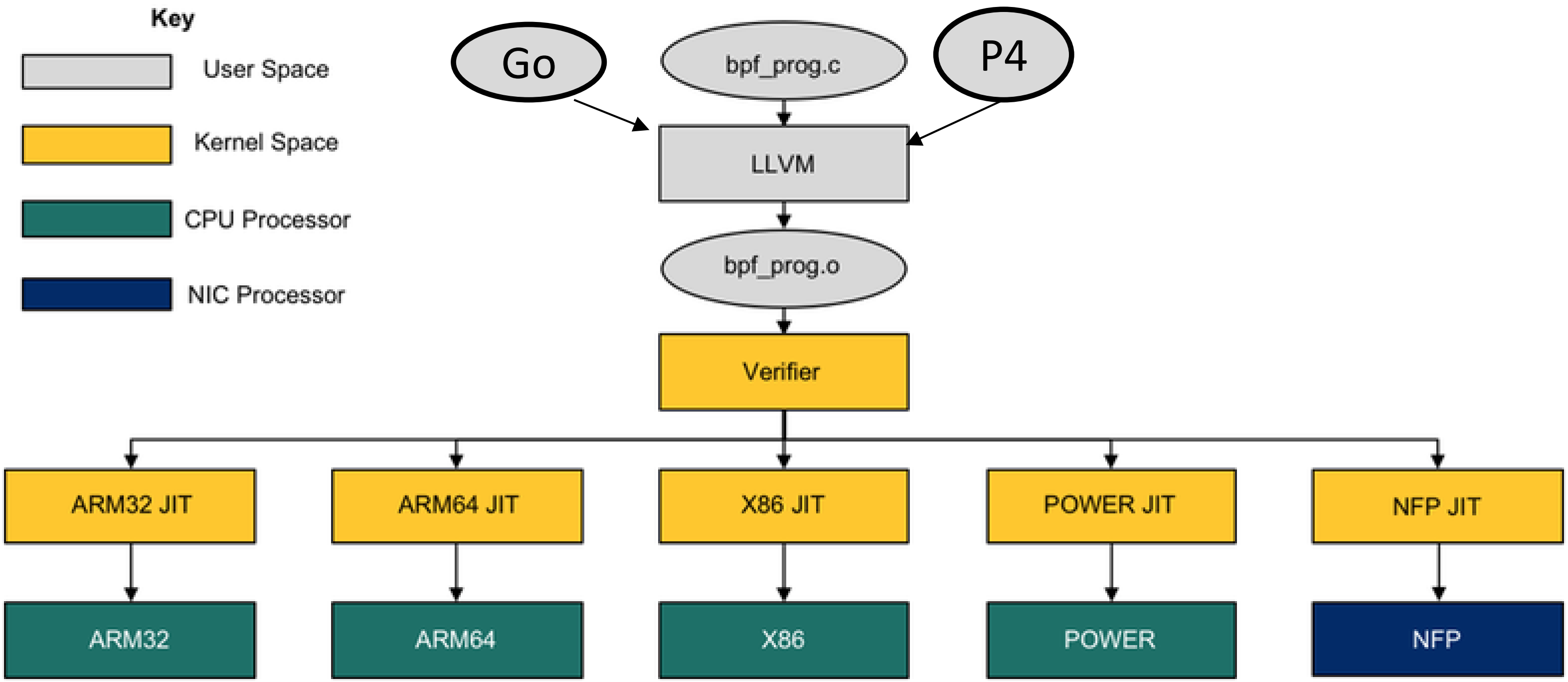
LD/STO

mode:3	size:2	insn_class:3
BPF_H	BPF_ABS	BPF_LD

eBPF Registers

- R0 : return value from function, and exit value for eBPF program
- R1 - R5 : arguments from eBPF program function
- R6 - R9 : callee saved registers that function preserve
- R10 - read-only frame pointer to access stack

Workflow


Key
User Space
Kernel Space
CPU Processor
NIC Processor
Go
bpf_prog.c
P4
LLVM
bpf_prog.o
Verifier
ARM32 JIT
ARM64 JIT
X86 JIT
POWER JIT
NFP JIT
ARM32
ARM64
X86
POWER
NFP

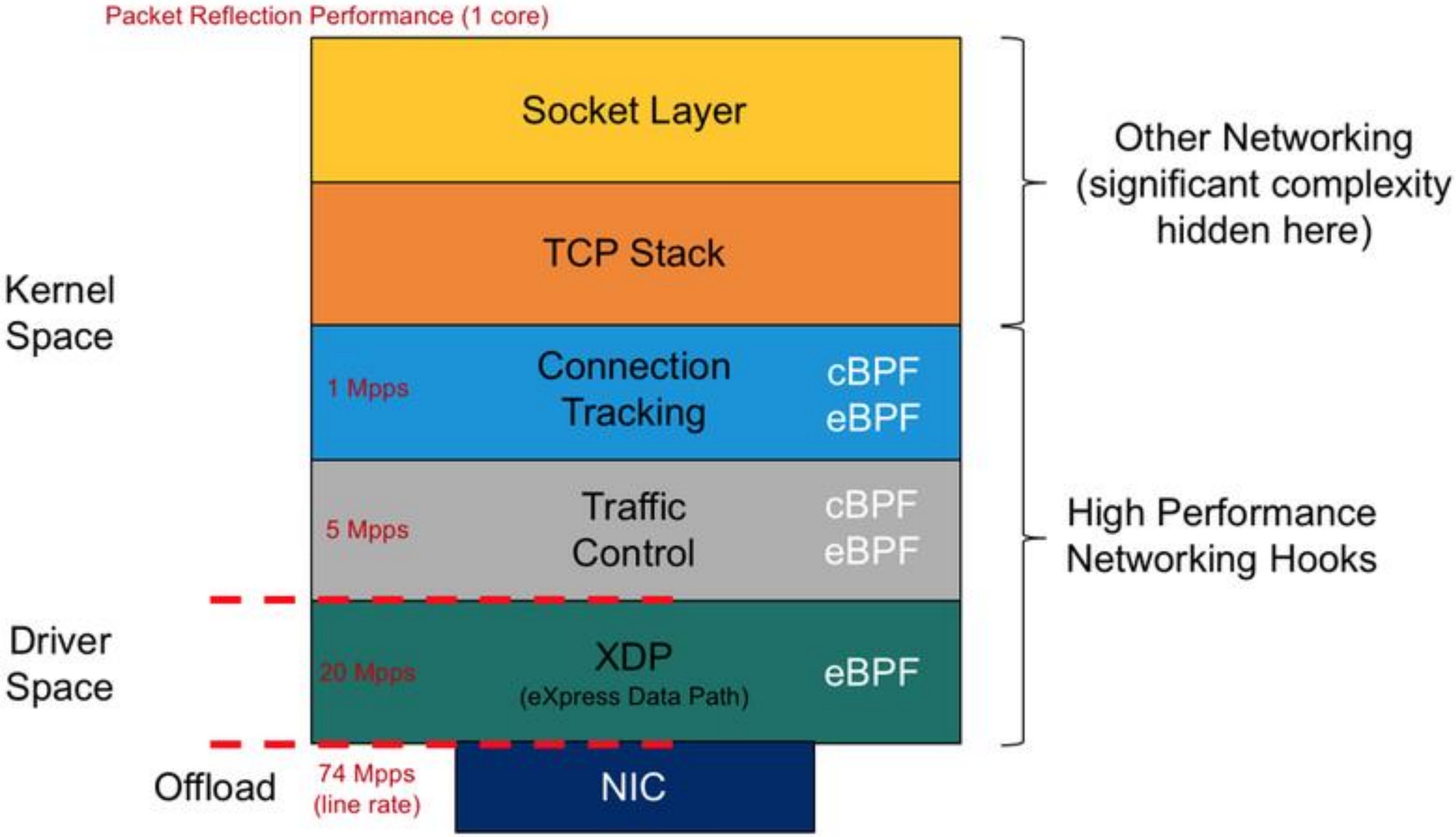
Restricted C for eBPF

- BPF has slightly different environment for C
- Subset of libraries (e.g. No printf())
- Helper functions and program context available
- Library functions all get inlined, no notion of function calls (yet)
- No global variables (use Maps)
- No loops (yet) unless unrolled by pragma or w/o verifier
- No const strings or data structures
- LLVM built-in functions usually available and inlined
- Partitioning processing path with tail calls
- Limited stack space up to 512 bytes

Hooks

- Code that handles intercepted function calls, events or messages between software components.
- Allows for user space applications to bypass the networking stack

Hooks

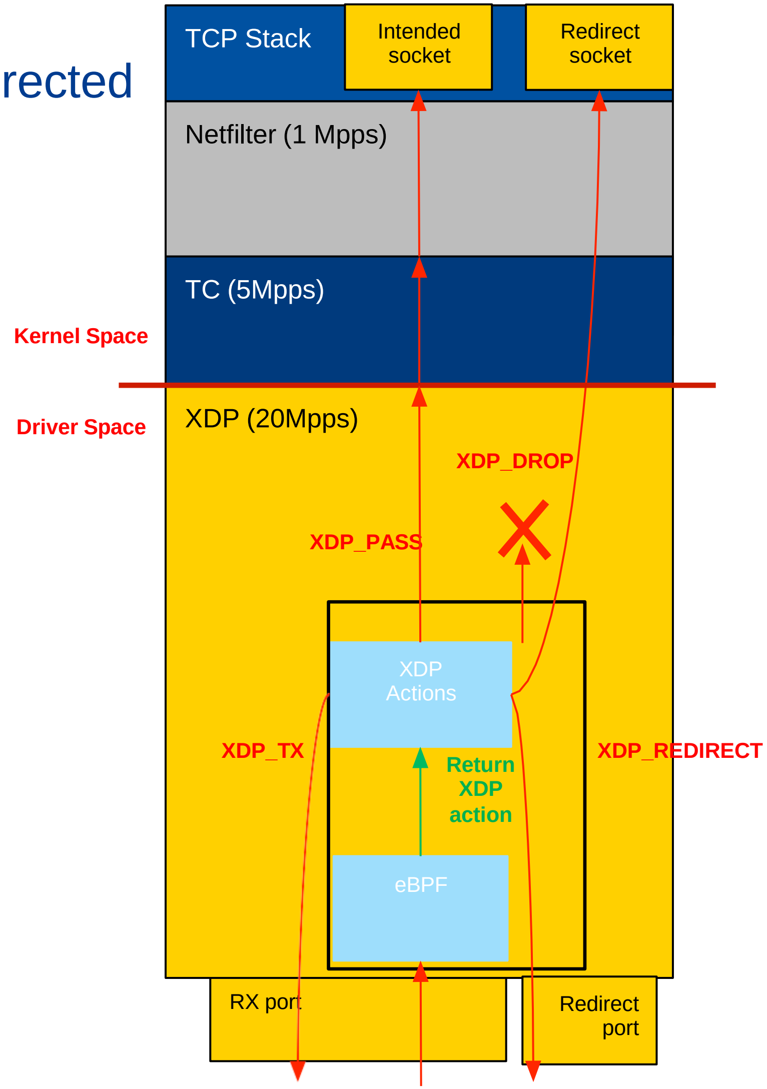

Packet Reflection Performance (1 core)
Socket Layer
TCP Stack
Other Networking (significant complexity hidden here)
Kernel Space
1 Mpps
Connection Tracking
cBPF
eBPF
5 Mpps
Traffic Control
cBPF
eBPF
High Performance Networking Hooks
Driver Space
20 Mpps
XDP
(eXpress Data Path)
eBPF
Offload
74 Mpps
(line rate)
NIC

What is XDP?

XDP allows packets to be reflected, filtered or redirected

without traversing networking stack

- eBPF programs classify/modify traffic and return XDP actions
  Note: cls_bpf in TC works in same manner
- XDP Actions
  - XDP_PASS
  - XDP_DROP
  - XDP_TX
  - XDP_REDIRECT
  - XDP_ABORT - Something went wrong
- Currently hooks onto RX path only
  - Other hooks can also work on TX



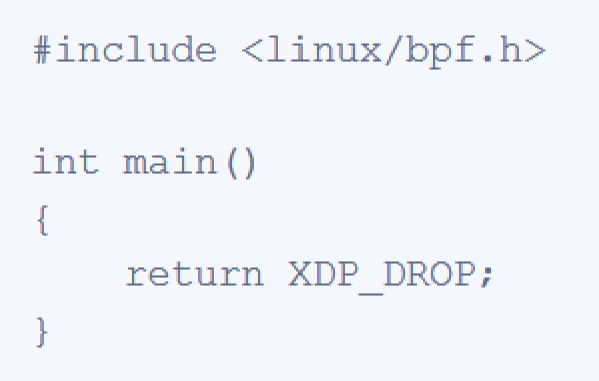
XDP Actions

Register 0 denotes the return value

Value	Action	Description
0	XDP_ABORTED	Error, Block the packet
1	XDP_DROP	Block the packet
2	XDP_PASS	Allow packet to continue up to the kernel
3	XDP_TX	Bounce the packet

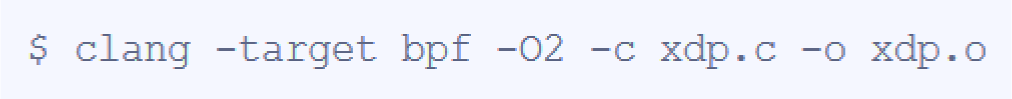
Hook example

- 1) Write C code:



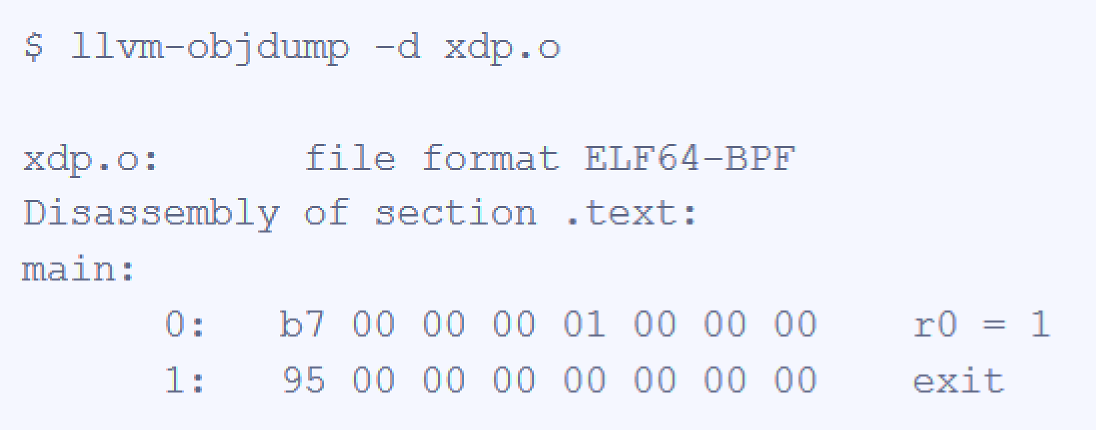
```
#include <linux/bpf.h>

int main()
{
    return XDP_DROP;
}
```

- 2) Compile to target BPF



```
$ clang -target bpf -O2 -c xdp.c -o xdp.o
```

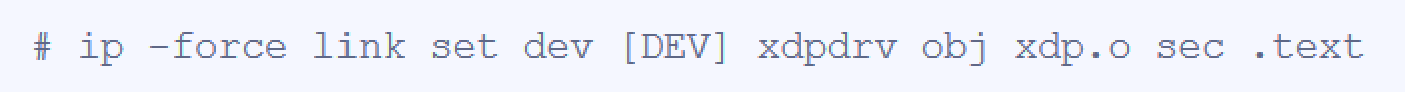
- Object generated:



```
$ llvm-objdump -d xdp.o

xdp.o:      file format ELF64-BPF
Disassembly of section .text:
main:
       0:    b7 00 00 00 01 00 00 00     r0 = 1
       1:    95 00 00 00 00 00 00 00     exit
```

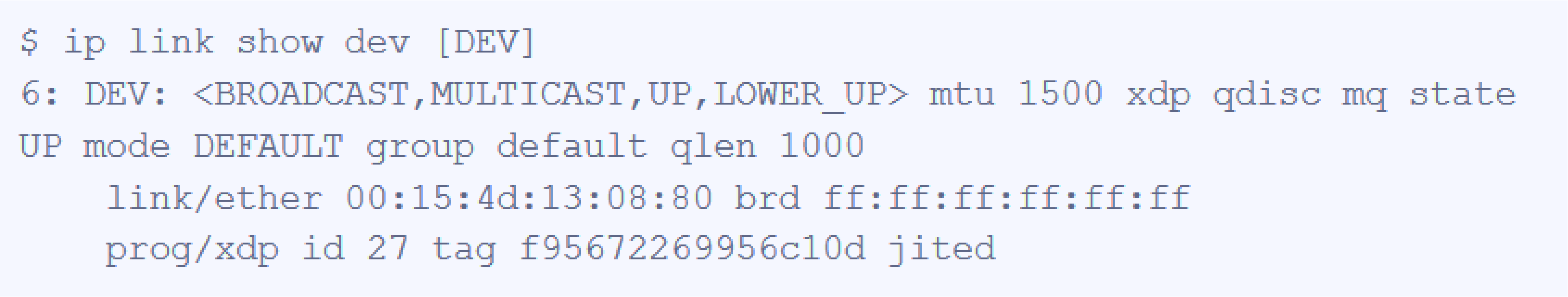
Hook example (2)

- 3) Load hook:



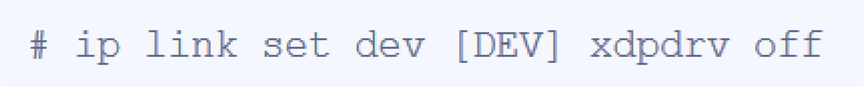
```
# ip -force link set dev [DEV] xdpdrv obj xdp.o sec .text
```

- Status:



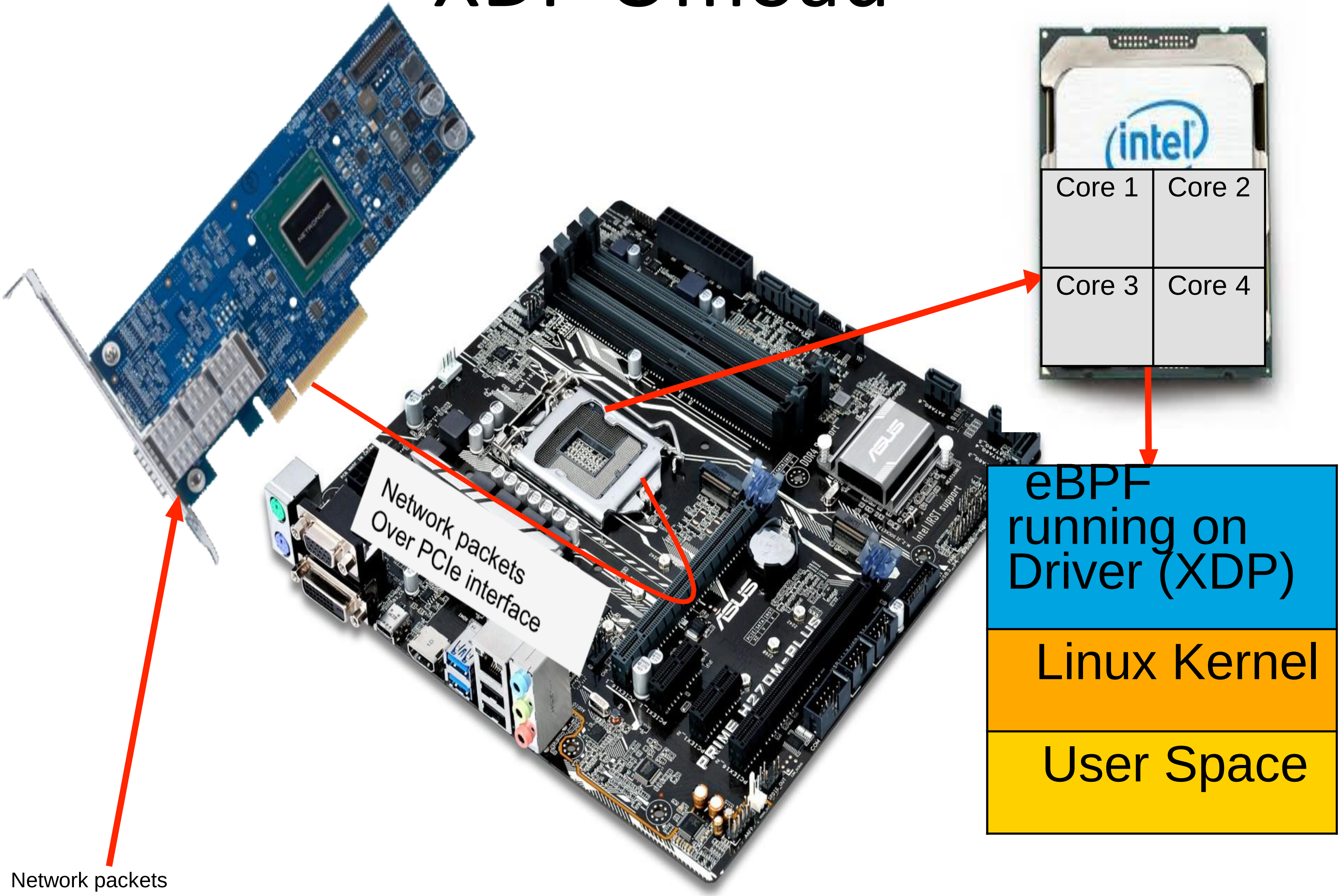
```
$ ip link show dev [DEV]
6: DEV: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc mq state
UP mode DEFAULT group default qlen 1000
    link/ether 00:15:4d:13:08:80 brd ff:ff:ff:ff:ff:ff
    prog/xdp id 27 tag f95672269956c10d jited
```

- 4) Unload:



```
# ip link set dev [DEV] xdpdrv off
```

XDP Offload



Network packets

Network packets Over PCIe interface

Core 1

Core 2

Core 3

Core 4

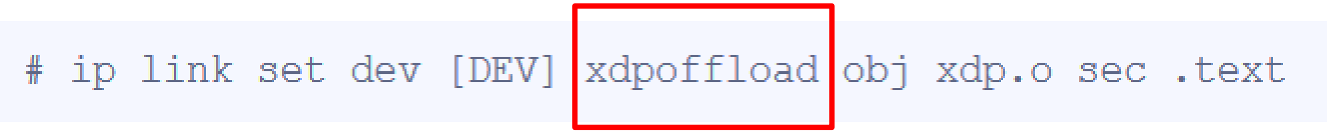
eBPF running on Driver (XDP)

Linux Kernel

User Space

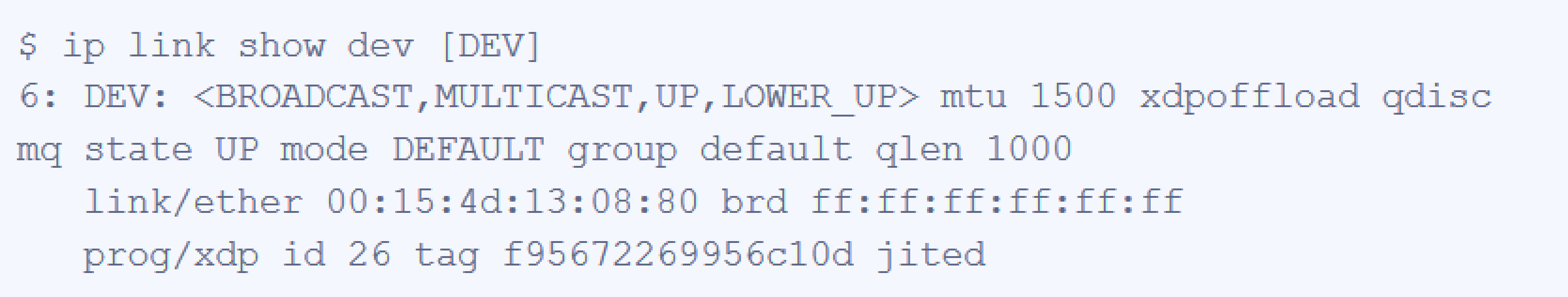
Hook example (3) - Offload

- 3) Offload:



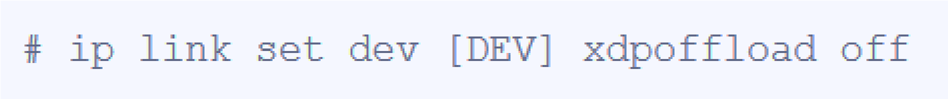
```
# ip link set dev [DEV] xdpoffload obj xdp.o sec .text
```

- Status:



```
$ ip link show dev [DEV]
6: DEV: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdpoffload qdisc
mq state UP mode DEFAULT group default qlen 1000
   link/ether 00:15:4d:13:08:80 brd ff:ff:ff:ff:ff:ff
   prog/xdp id 26 tag f95672269956c10d jited
```

- 4) Unload:

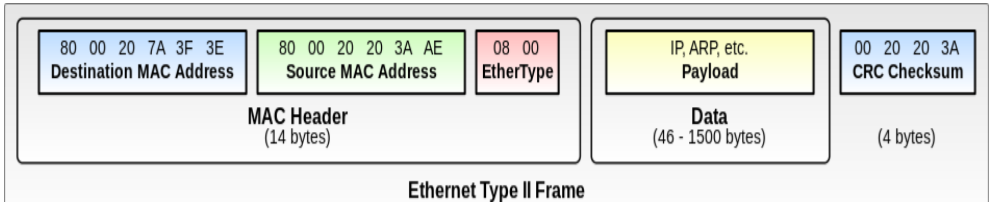


```
# ip link set dev [DEV] xdpoffload off
```

eBPF example

Drop packets not EtherType 0x2222

```
#include <linux/bpf.h>
#include "bpf_api.h"
#include "bpf_helpers.h"

SEC(“xdp_prog1”)
int xdp_prog1(struct xdp_md *xdp)
{
        unsigned char *data;

        data = (void *)(unsigned long)xdp->data;
        if (data + 14 > (void *)(long)xdp->data_end)
                return XDP_ABORTED;

        if (data[12] != 0x22 || data[13] != 0x22)
                return XDP_DROP;

        return XDP_PASS;
}
```



Clang Compiler

```
xdp_prog1:
        0:       b7 00 00 00 00 00 00 00       r0 = 0
        1:       61 12 04 00 00 00 00 00       r2 = *(u32 *)(r1 + 4)
        2:       61 11 00 00 00 00 00 00       r1 = *(u32 *)(r1 + 0)
        3:       bf 13 00 00 00 00 00 00       r3 = r1
        4:       07 03 00 00 0e 00 00 00       r3 += 14


        5:       2d 23 07 00 00 00             if r3 > r2
        6:       00 00                         goto 7 r0 =
                 b7 00 00 00 01 00             1
                 00 00
        7:       71 12 0c 00 00 00             r2 = *(u8 *)(r1
        8:       00 00                         + 12) if r2 !=
                 55 02 04 00 22 00             34 goto 4
                 00 00
        9:       71 11 0d 00 00 00             r1 = *(u8 *)(r1
                 00 00                         + 13)
LBB0_4:
       13:       95 00 00 00 00 00 00 00       exit
```

Maps

Maps are key-value stores used to store state

- Up to 128 maps per program
- Infinite size
- Multiple different types-Non XDP
  - 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
  - BPF_MAP_TYPE_LPM_TRIE
  - BPF_MAP_TYPE_ARRAY_OF_MAPS
  - BPF_MAP_TYPE_HASH_OF_MAPS
  - BPF_MAP_TYPE_DEVMAP
  - BPF_MAP_TYPE_SOCKMAP
  - BPF_MAP_TYPE_CPUMAP
- Accessed via map helpers

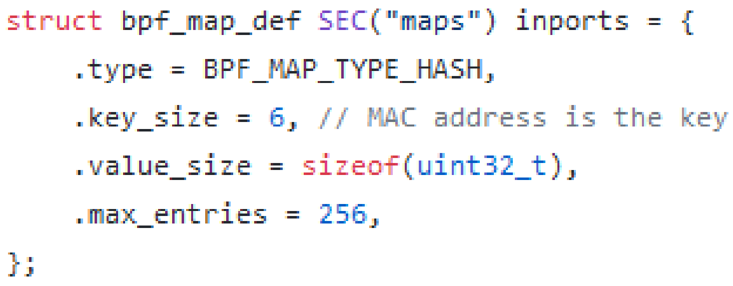
Key	Value
0	10.0.0.1
19	10.0.0.6
91	10.0.1.1
4121	121.0.0.1
12111	5.0.2.12
...	...

Maps types

- BPF_MAP_TYPE_HASH: a hash table
- BPF_MAP_TYPE_ARRAY: an array map, optimized for fast lookup speeds, often used for counters
- BPF_MAP_TYPE_PROG_ARRAY: an array of file descriptors corresponding to eBPF programs; used to implement jump tables and sub-programs to handle specific packet protocols
- BPF_MAP_TYPE_PERCPU_ARRAY: a per-CPU array, used to implement histograms of latency
- BPF_MAP_TYPE_PERF_EVENT_ARRAY: stores pointers to struct perf_event, used to read and store perf event counters
- BPF_MAP_TYPE_CGROUP_ARRAY: stores pointers to control groups
- BPF_MAP_TYPE_PERCPU_HASH: a per-CPU hash table
- BPF_MAP_TYPE_LRU_HASH: a hash table that only retains the most recently used items
- BPF_MAP_TYPE_LRU_PERCPU_HASH: a per-CPU hash table that only retains the most recently used items
- BPF_MAP_TYPE_LPM_TRIE: a longest-prefix match trie, good for matching IP addresses to a range
- BPF_MAP_TYPE_STACK_TRACE: stores stack traces
- BPF_MAP_TYPE_ARRAY_OF_MAPS: a map-in-map data structure
- BPF_MAP_TYPE_HASH_OF_MAPS: a map-in-map data structure
- BPF_MAP_TYPE_DEVICE_MAP: for storing and looking up network device references
- BPF_MAP_TYPE_SOCKET_MAP: stores and looks up sockets and allows socket redirection with BPF helper functions

Maps

- The map is defined by:
  - Type
  - key size in bytes
  - value size in bytes
  - max number of elements



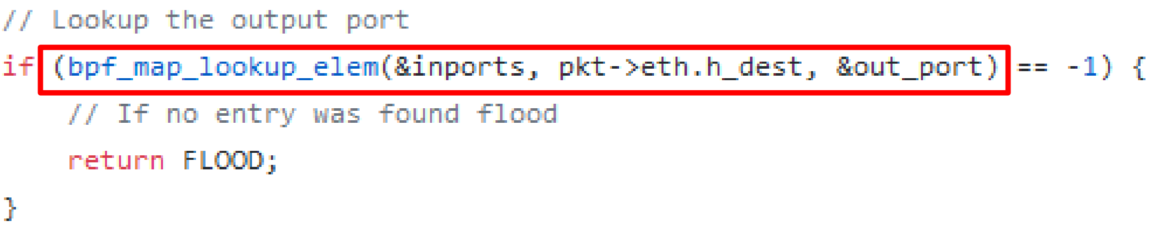
```
struct bpf_map_def SEC("maps") inports = {
    .type = BPF_MAP_TYPE_HASH,
    .key_size = 6, // MAC address is the key
    .value_size = sizeof(uint32_t),
    .max_entries = 256,
};
```

Key (MAC address)	Value (output port number)
0123456789AB	6
CAFEDEADFF	1
…	…

Helpers

Helpers are used to add functionality that would otherwise be difficult

- Key XDP Map helpers
  - bpf_map_lookup_elem
  - bpf_map_update_elem
  - bpf_map_delete_elem
  - bpf_redirect_map
- Head Extend
  - bpf_xdp_adjust_head
  - bpf_xdp_adjust_meta
- Others
  - bpf_ktime_get_ns
  - bpf_trace_printk
  - bpf_tail_call
  - Bpf_redirect



```
// Lookup the output port
if (bpf_map_lookup_elem(&inports, pkt->eth.h_dest, &out_port)) == -1) {
    // If no entry was found flood
    return FLOOD;
}
```

https://github.com/torvalds/linux/blob/master/include/uapi/linux/bpf.h

Open Source Tools

Bpftool

- Lists active bpf programs and maps
- Interactions with eBPF maps (lookups or updates)
- Dump assembly code (JIT and Pre-JIT)

Iproute2

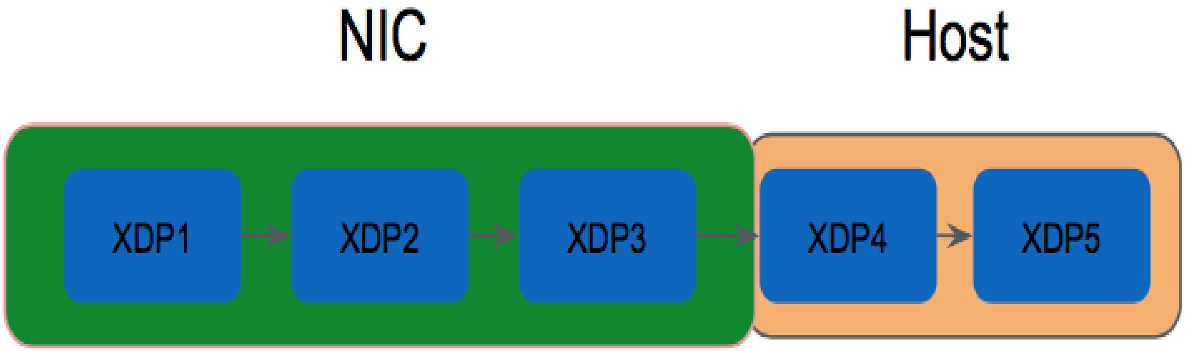
- Can load and attach eBPF programs to TC, XDP or XDP offload (SmartNIC)

Libbpf

- BPF library allowing for user space program access to eBPF api

Kernel Offload - Multi-Stage Processing

- Use of offloads does not preclude standard in-driver XDP use
- Offload some programs, leave some running on the host
- Maximize efficiency by playing to NFPs and host’s strengths
- Communication between programs via XDP/SKB metadata



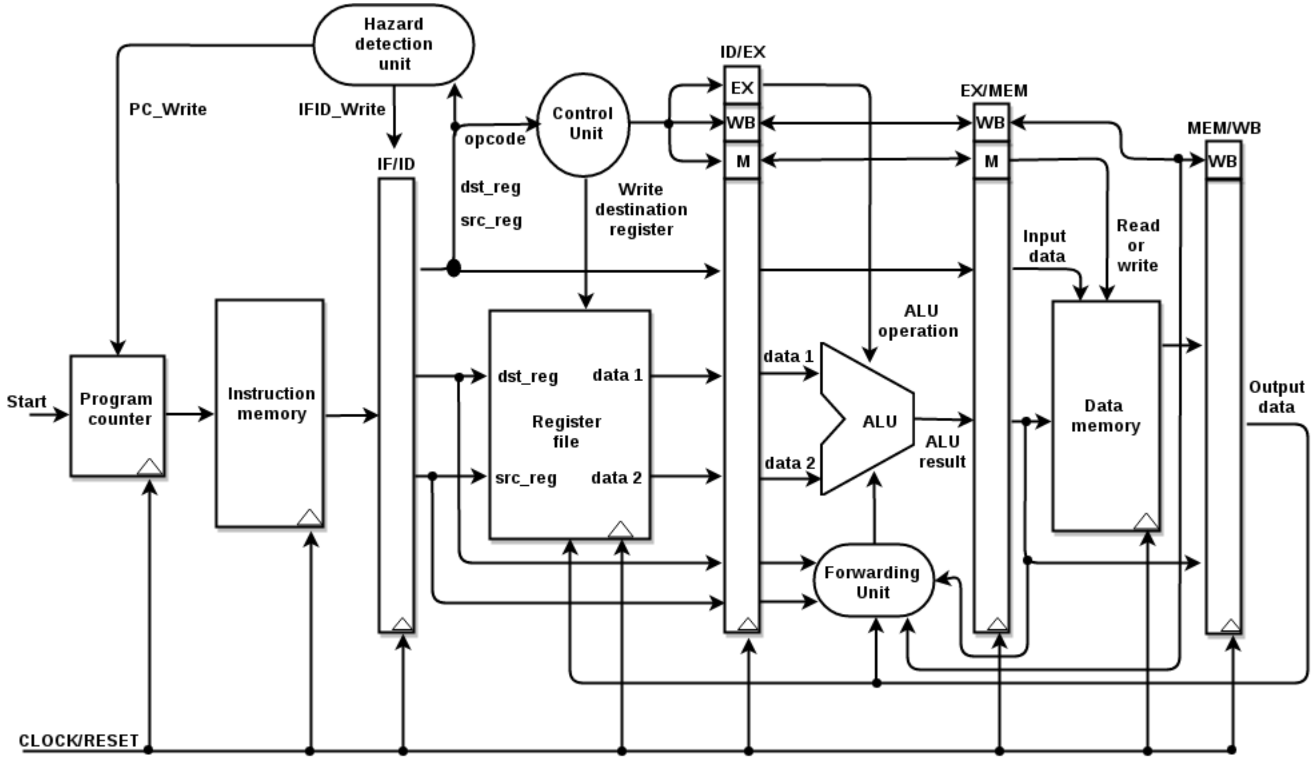
Use Cases

- Load Balacing
- DDoS mitigation
- Monitoring
- Distributed Firewall
- Intruction Detection System
- NIC Behavior (Receive Side Scaling)

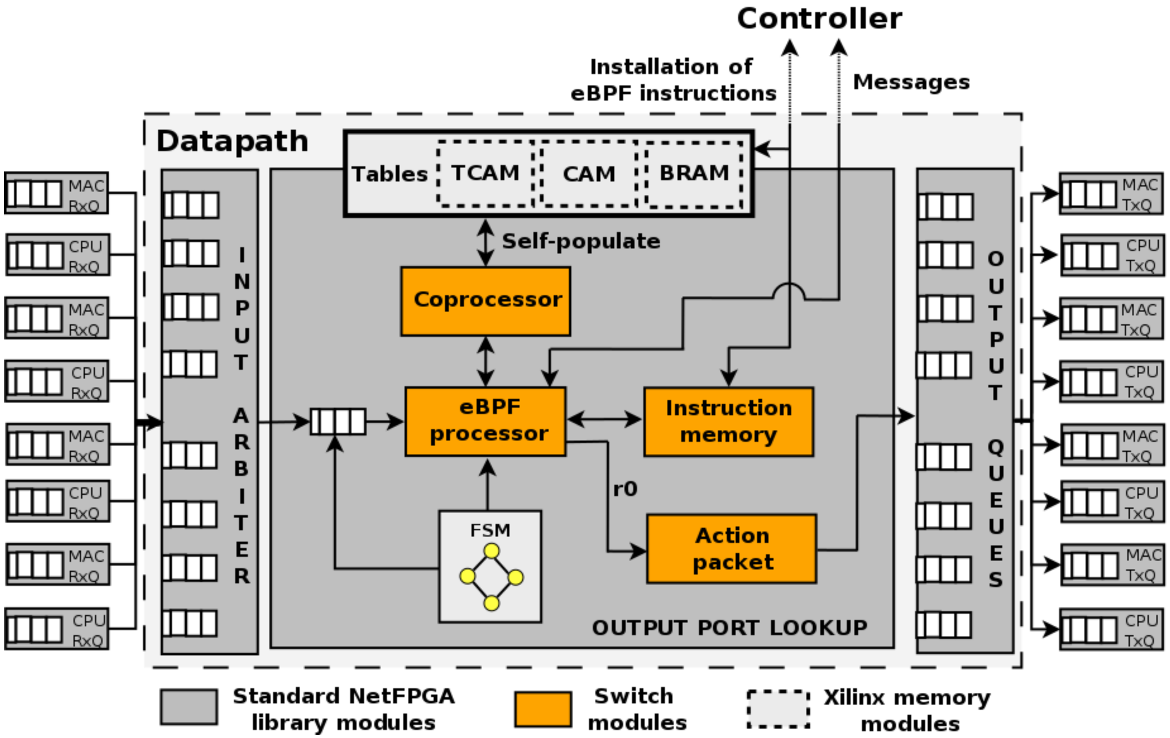
Projects

Layer	Hardware	Software
NIC	Netronome	XDP/Kernel
Switch	Developing a project with NetFPGA-SUME	BPFabric

A Programmable Protocol-Independent Hardware Switch with Dynamic Parsing, Matching, and Actions



A Programmable Protocol-Independent Hardware Switch with Dynamic Parsing, Matching, and Actions



P4 limitations

- ▪P4-14 has some essential restrictions.
  - If-else statement can only be used in the control block.
  - It does not support for-loop.
  - It has only a limited set of primitive actions.

- P4 to eBPF
- https://github.com/iovisor/bcc/tree/master/src/cc/frontends/p4

Why is eBPF cool?

- You can do whatever you want
  - E.g. sketches (telemetry)
  - Timers (Management)
- Program in C, P4
- Change in real-time

Conclusions

- Fast (relatively) easy to use, potentially very powerful
  - Monitoring and (likely) network processing
- Many use cases
  - Packet filters (copy packet and pass to user space)
    - Used by tcpdump/libpcap, wireshark, nmap, dhcp, arpd, ...
  - In-kernel networking subsystems
    - cls_bpf (TC classifier) – QoS subsystem- , xt_bpf, ppp,...
  - seccomp (chrome sandboxing)
    - Introduced in 2012 to filter syscall arguments with bpf program
  - Tracing, Networking, Security, …
- Several “big names” here
- Need to enlarge the community, particularly with respect to
- end-users and application (e.g., non-kernel) developers

Join us

- mmvieira@dcc.ufmg.br

Kernel Security and Stability

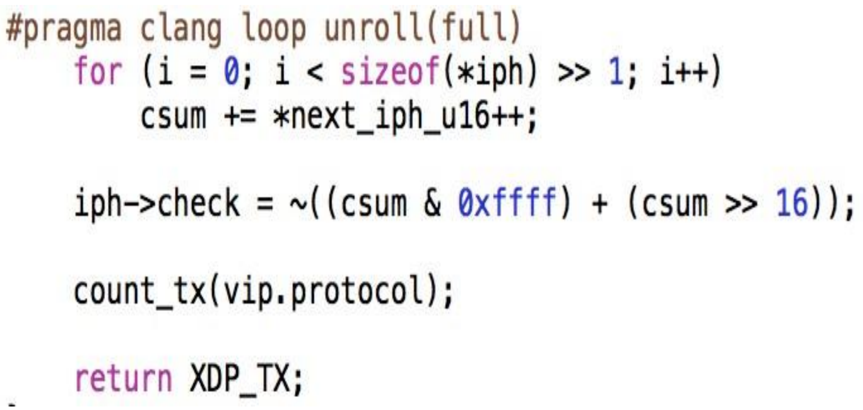
eBPF code injected into the kernel must be safe

- ▸ Potential risks
  - Infinite loops could crash the kernel
  - Buffer overflows
  - Uninitialized variables
  - Large programs may cause performance issues
  - Compiler errors

eBPF Verifier

The verifier checks for the validity of programs

- Ensure that no back edges (loops) exist
  - Mitigated through the use #pragma unroll
- Ensure that the program has no more than 4,000 instructions
- There are also a number of other checks on the validity of register usage
  - These are done by traversing each path through the program
- If there are too many possible paths the program will also be rejected
  - 1K branches
  - 130K complexity of total instructions

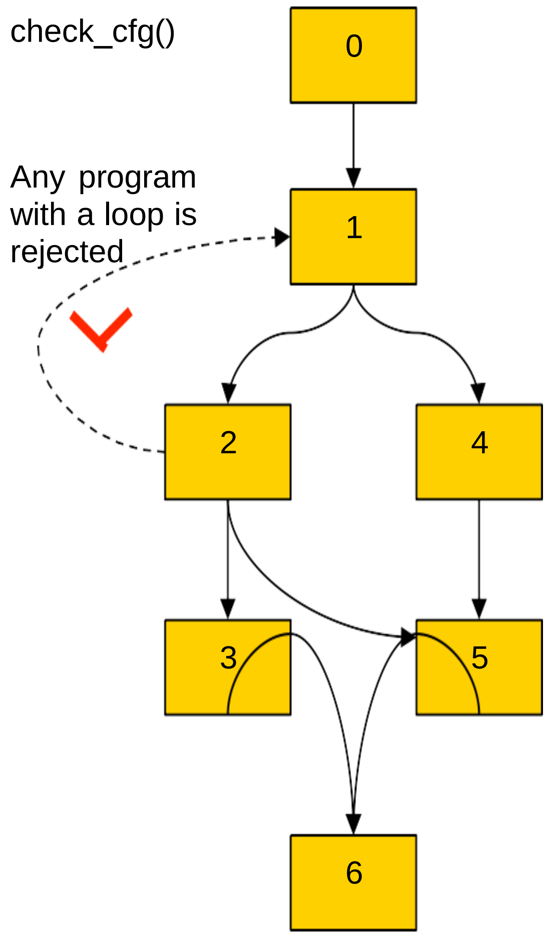


```
#pragma clang loop unroll(full)
    for (i = 0; i < sizeof(*iph) >> 1; i++)
        csum += *next_iph_u16++;

    iph->check = ~((csum & 0xffff) + (csum >> 16));

    count_tx(vip.protocol);

    return XDP_TX;
```

eBPF Verifier

The verifier checks for the DAG property

- Ensures that no back edges (loops) exist
- Backward jumps are allowed
  - Only if they do not cause loops
- Handled by check_cfg() in verifier.c



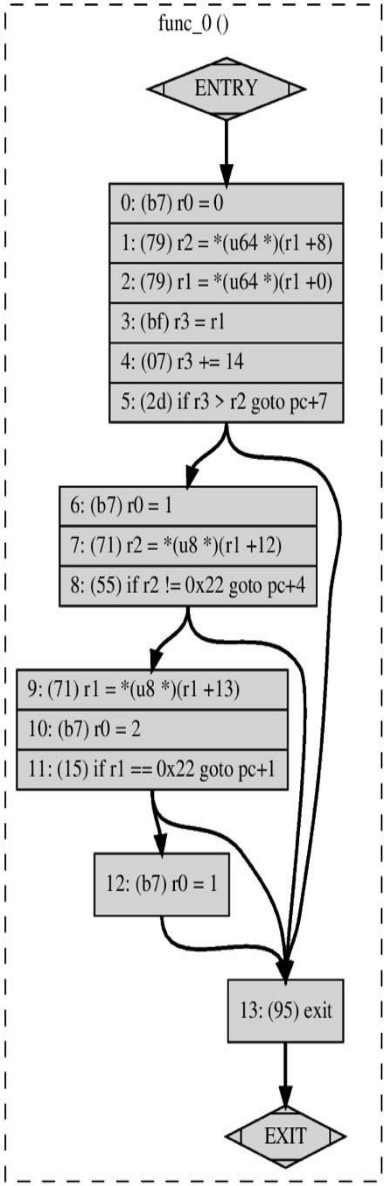
check_cfg()

Any program with a loop is rejected

DAG

```
#include <linux/bpf.h>
#include "bpf_api.h"
#include "bpf_helpers.h"

SEC(“xdp_prog1”)
int xdp_prog1(struct xdp_md *xdp)
{
        unsigned char *data;

        data = (void *)(unsigned long)xdp->data;
        if (data + 14 > (void *)(long)xdp->data_end)
                return XDP_ABORTED;

        if (data[12] != 0x22 || data[13] != 0x22)
                return XDP_DROP;

        return XDP_PASS;
}
```

```
xdp_prog1:
            r0 = 0
            r2 = *(u32 *)(r1
+ 4)
            r1 = *(u32 *)(r1
+ 0)
            r3 = r1
            r3 += 14
            if r3 > r2 goto 7
            r0 = 1
            r2 = *(u8 *)(r1 +
12)
            if r2 != 34 goto
4
            r1 = *(u8 *)(r1 +
13)
            r0 = 2
            if r1 == 34 goto
1
            r0 = 1
```



```
0: (b7) r0 = 0
1: (79) r2 = *(u64 *)(r1 +8)
2: (79) r1 = *(u64 *)(r1 +0)
3: (bf) r3 = r1
4: (07) r3 += 14
5: (2d) if r3 > r2 goto pc+7
6: (b7) r0 = 1
7: (71) r2 = *(u8 *)(r1 +12)
8: (55) if r2 != 0x22 goto pc+4
9: (71) r1 = *(u8 *)(r1 +13)
10: (b7) r0 = 2
11: (15) if r1 == 0x22 goto pc+1
12: (b7) r0 = 1
13: (95) exit
```

DAG shown with bpftool and dot graph generator

```
  # bpftool prog dump xlated id 13 visual > cfg.txt
  # dot -Tps cfg.txt -o cfg.ps
```

What is Berkeley Packet Filter (BPF)?

- tcpdump -i eno1 –ddd IPv4

```
12
40 0 0 12
21 0 2 2048
48 0 0 23
21 6 7 1
21 0 6 34525
48 0 0 20
21 3 0 58
21 0 3 44
48 0 0 54
21 0 1 58
6 0 0 262144
6 0 0 0
```

Original use-case: tcpdump filter for raw packet sockets

What is BPF?

- tcpdump -i eno1 -d icmp or icmp6

```
(000) ldh [12]                    #ethertype field
(001) jeq #0x800 jt 2 jf 4    # IPv4?
(002) ldb [23]
(003) jeq #0x1 jt 10 jf 11    # ICMP==1?
(004) jeq #0x86dd jt 5 jf 11  #IPv6?
(005) ldb [20]
(006) jeq #0x3a jt 10 jf 7  # ICMPv6==58?
(007) jeq #0x2c jt 8 jf 11  # IPv6-Frag
(008) ldb [54]
(009) jeq #0x3a jt 10 jf 11  # ICMPv6
(010) ret #262144
(011) ret #0
```