eBPF/XDP

SIGCOMM 2018

David Beckett
Jakub Kicinski

NETRONOME

Jakub Kicinski

Lead Software Engineer

eBPF Kernel Development

David Beckett

Software Engineer

eBPF Application Development

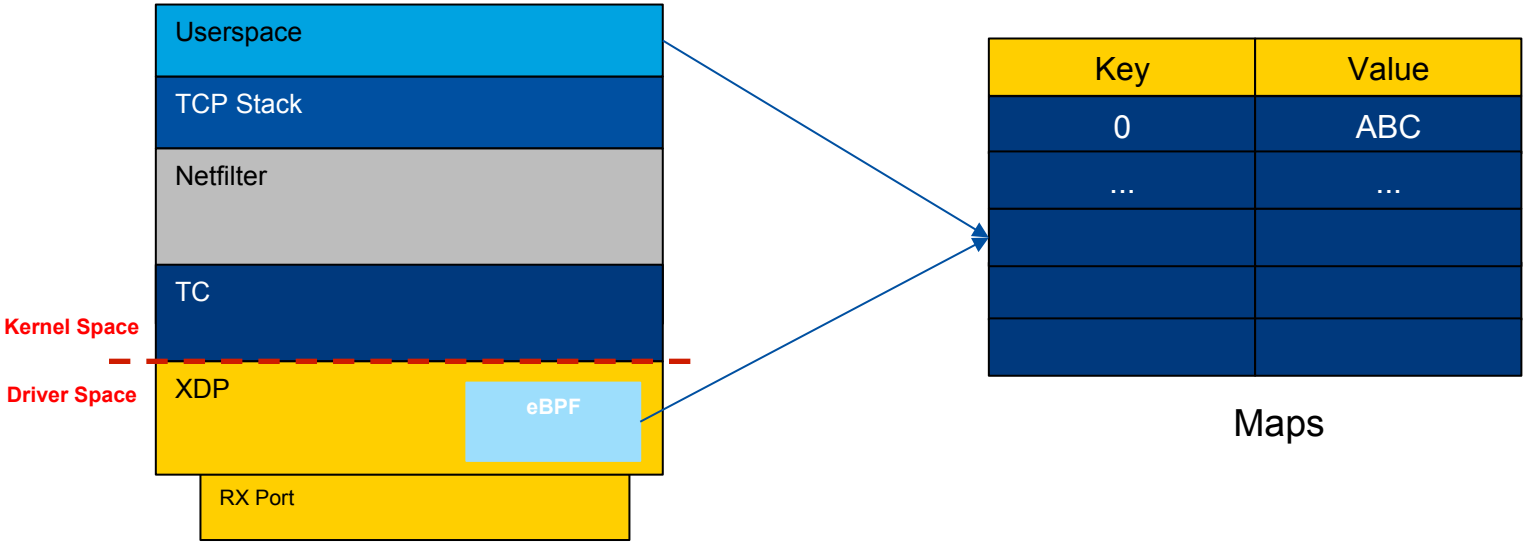
NETRONOME

Overview

NETRONOME

- What is eBPF/XDP?
- Demos
- SmartNIC eBPF offload
- Host dataplane Acceleration
- SmartNIC offload Demos

eBPF System

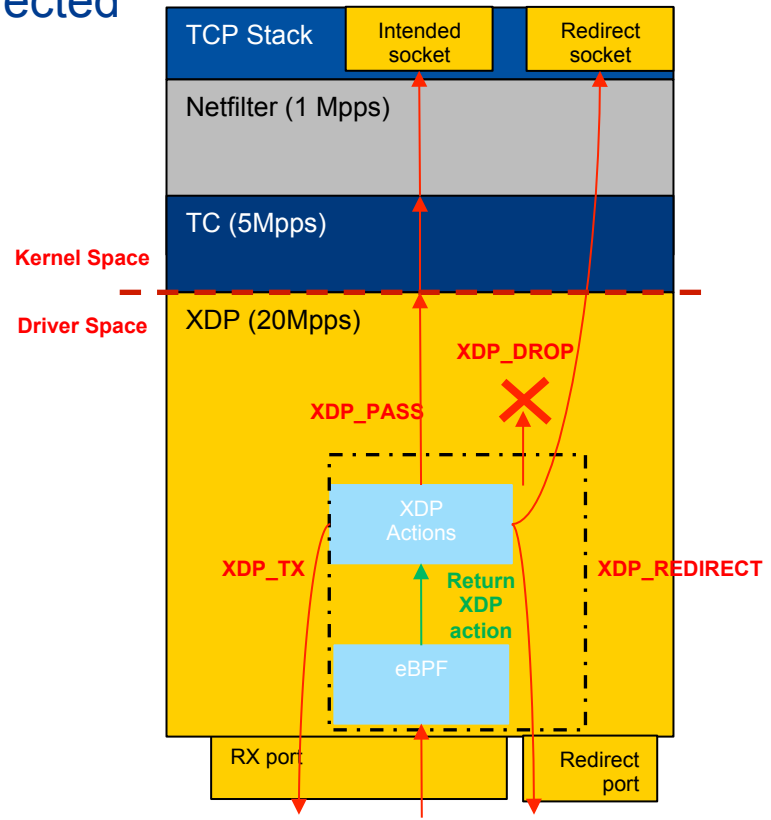


What is XDP?

NETRONOME

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

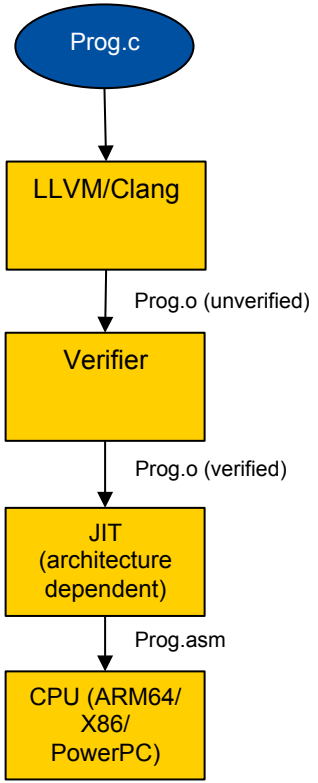


What is the eBPF Architecture?

NETRONOME

A kernel-based virtual machine to enable low-level packet processing

- Think Java VMs in the kernel
  - Networking focused ISA/bytecode
  - 10 64-bit registers
    - 32-bit subregisters
  - Small stack (512 bytes)
  - Infinite-size key value stores (maps)
- Write programs in C, P4, Go or Rust
  - C is LLVM compiled to BPF bytecode
  - Verifier checked
  - JIT converts to assembly
- Hooks into the kernel in many places
  - Final packet handling dependent on hook



NETRONOME

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
...	...

Helpers are used to add functionality that would otherwise be difficult

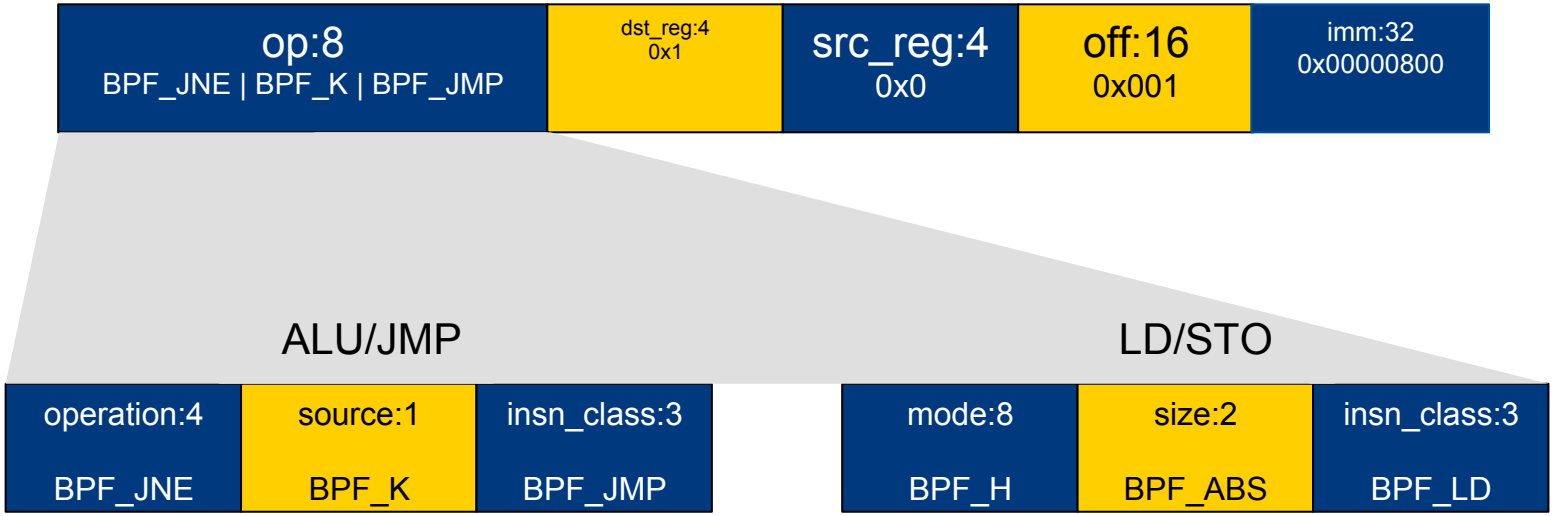
- Key XDP Map helpers
  - bpf_map_lookup_elem
  - bpf_map_update_elem
  - bpf_map_delete_elem
  - bpf_redirect_map
- Head Extend
  - bpf_xdp_adjust_head
  - bpf_xdp_adjust_meta
- Others
  - bpf_ktime_get_ns
  - bpf_trace_printk
  - bpf_tail_call
  - Bpf_redirect

```
if (is_ipv6)
    memcpy(vip.daddr.v6, pckt.dstv6, 16);
else
    vip.daddr.v4 = pckt.dst;

vip.dport = pckt.port16[1];
vip.protocol = pckt.proto;
vip_info = bpf_map_lookup_elem(&vip_map, &vip);
if (!vip_info) {
    vip.dport = 0;
    vip_info = bpf_map_lookup_elem(&vip_map, &vip);
    if (!vip_info)
        return XDP_DROP;
    pckt.port16[1] = 0;
}
```

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

NETRONOME

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



XDP Actions

NETRONOME

Register 0 denotes the return value

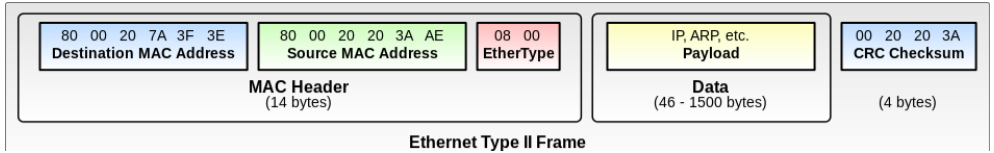
Value	Action	Description
0	XDP_ABORTED	Error, Block the packet
1	XDP_DROP	Block the packet
2	XDP_PASS	Allow packet to continue up to the kernel
3	XDP_TX	Bounce the packet

Code Snippet - XDP/eBPF Example

NETRONOME

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 00 00       if r3 > r2 goto 7
       6:       b7 00 00 00 01 00 00 00       r0 = 1
       7:       71 12 0c 00 00 00 00 00       r2 = *(u8 *)(r1 + 12)
       8:       55 02 04 00 22 00 00 00       if r2 != 34 goto 4
       9:       71 11 0d 00 00 00 00 00       r1 = *(u8 *)(r1 + 13)
      10:       b7 00 00 00 02 00 00 00       r0 = 2
      11:       15 01 01 00 22 00 00 00       if r1 == 34 goto 1
      12:       b7 00 00 00 01 00 00 00       r0 = 1

LBB0_4:
      13:       95 00 00 00 00 00 00 00       exit
```

Kernel Security and Stability

NETRONOME

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

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

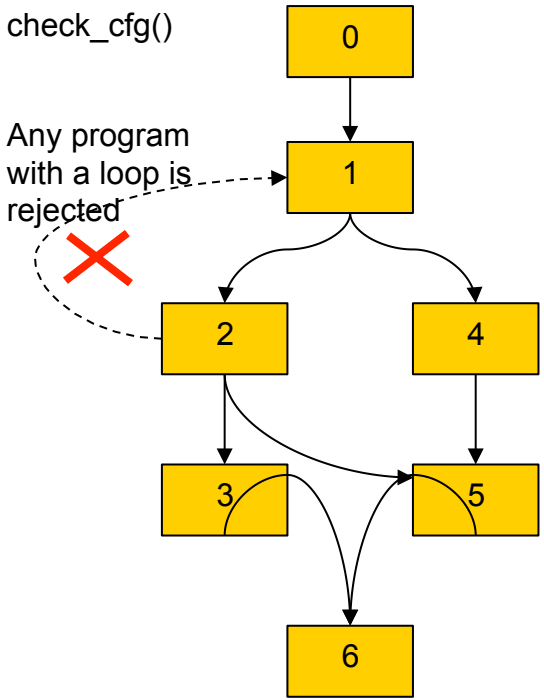
```
#pragma clang loop unroll(full)
    for (i = 0; i < sizeof(*iph) >> 1; i++)
        csum += *next_iph_u16++;

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

    count_tx(vip.protocol);

    return XDP_TX;
```

Verifier-Directed Acyclical Graph

NETRONOME

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



DAG Example

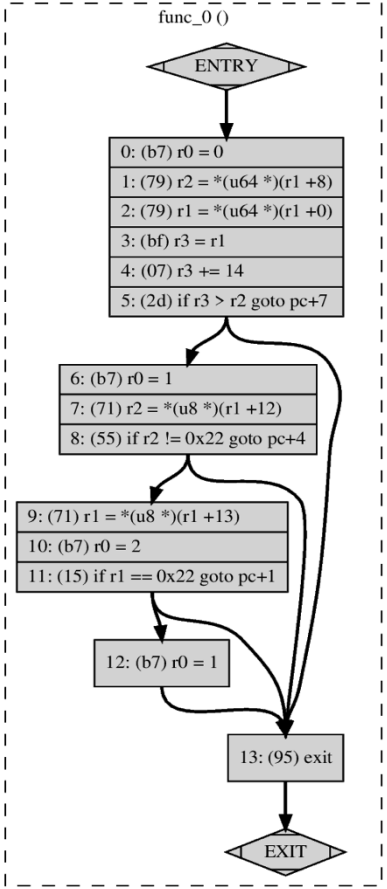
NETRONOME

```
#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;
}
```

DAG shown with bpftool and dot graph generator

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

```
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
```



x86 JIT Code - XDP/eBPF Example

NETRONOME

```
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
```

Verifier

JITed for x86 CPU

```
   0:              push   %rbp
   1:              mov    %rsp,%rbp
   4:              sub    $0x28,%rsp
   b:              sub    $0x28,%rbp
   f:              mov    %rbx,0x0(%rbp)
  13:              mov    %r13,0x8(%rbp)
  17:              mov    %r14,0x10(%rbp)
  1b:              mov    %r15,0x18(%rbp)
  1f:              xor    %eax,%eax
  21:              mov    %rax,0x20(%rbp)
  25:              xor    %eax,%eax
  27:              mov    0x8(%rdi),%rsi
  2b:              mov    0x0(%rdi),%rdi
  2f:              mov    %rdi,%rdx
  32:              add    $0xe,%rdx
  36:              cmp    %rsi,%rdx
  39:              ja
0x0000000000000060
  3b:              mov    $0x1,%eax
  40:              movzbq 0xc(%rdi),%rsi
  45:              cmp    $0x22,%rsi
  49:              jne
0x0000000000000060
  4b:              movzbq 0xd(%rdi),%rdi
  50:              mov    $0x2,%eax
  55:              cmp    $0x22,%rdi
  59:              je
0x0000000000000060
  5b:              mov    $0x1,%eax
  60:              mov    0x0(%rbp),%rbx
  64:              mov    0x8(%rbp),%r13
  68:              mov    0x10(%rbp),%r14
  6c:              mov    0x18(%rbp),%r15
  70:              add    $0x28,%rbp
  74:              leaveq
  75:              retq
```

Open Source Tools

NETRONOME

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

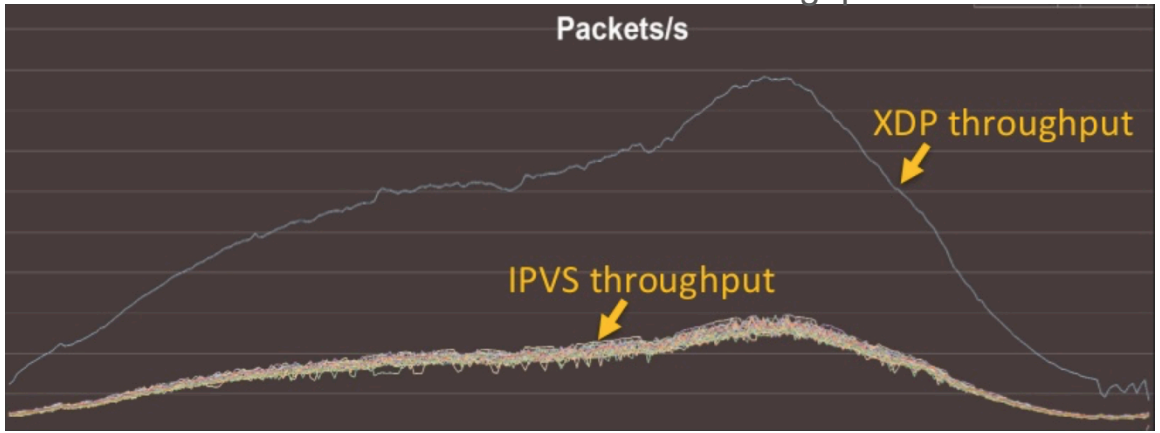
Public XDP Use Cases

NETRONOME

Current use cases focus on load balancers, DDoS mitigation and simple monitoring

- Load balancer
  - Used by FB Katran to replace IPVS - 2X performance per core
- DDoS mitigation
  - Cloudflare starting the transition to eBPF
- Distributed Firewall
  - Flexible, high-performance blacklisting

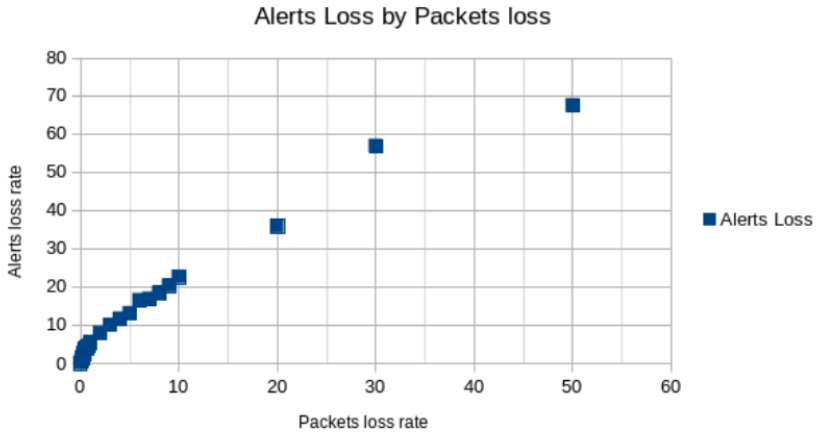
FB Load Balancer throughput: XDP vs IPVS

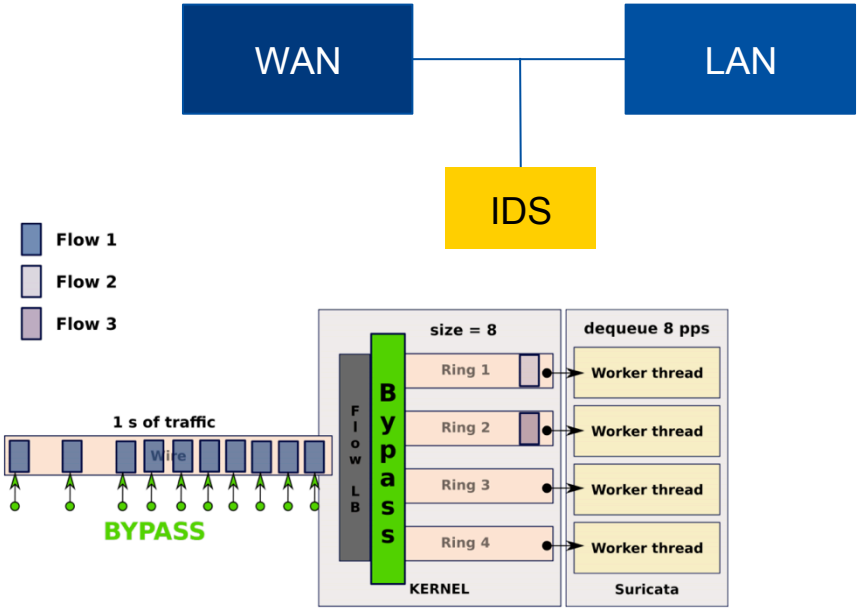


NETRONOME

Suricata Intrusion Detection System (IDS)

- Whitelist large flows (e.g. Netflix stream)





STAMUS NETWORKS

"Suricata Performance with a S like Security" É. Leblond

Summary: Driver XDP

NETRONOME

Advantages

- Increased performance - 4X
- Reuses kernel infrastructure
- Upstream-boot Linux and you are good to go
- Allows updates of low-level functionality without kernel reboot
  - This should not be underestimated
  - A particular DC provider spent 3 months rebooting servers when a bug was found

Disadvantages

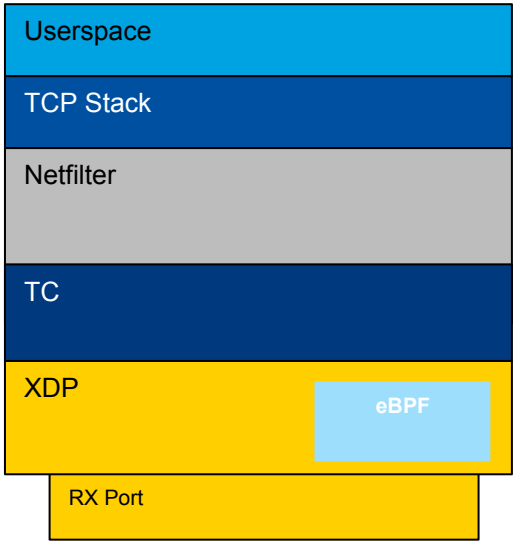
- CPU still limits the use-cases at high data rates

Demo 1 - XDP Actions and Packet Modification

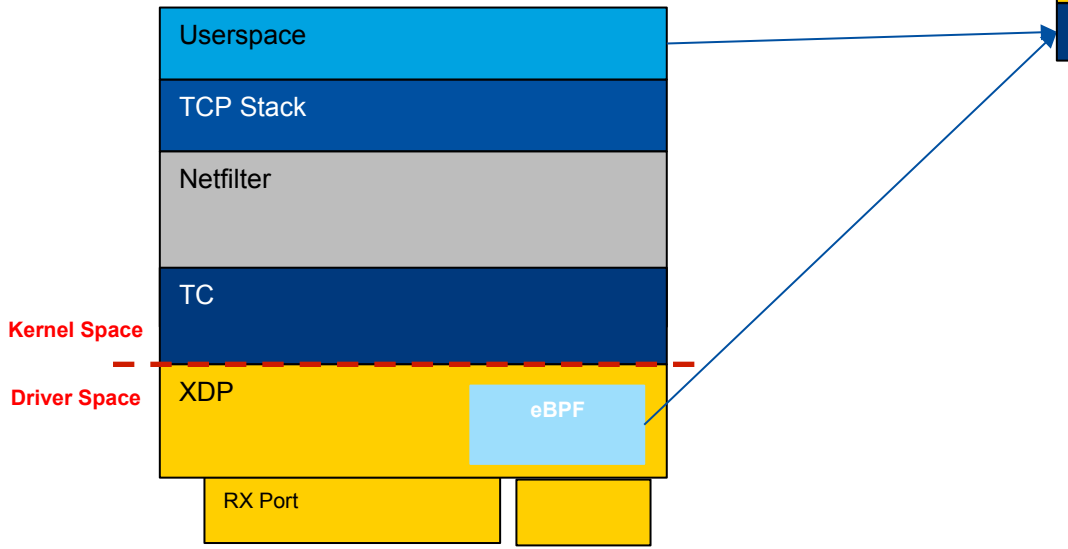
NETRONOME

xdp_drop

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

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



xdp_actions based on eBPF map



Key	Value
0	XDP_TX

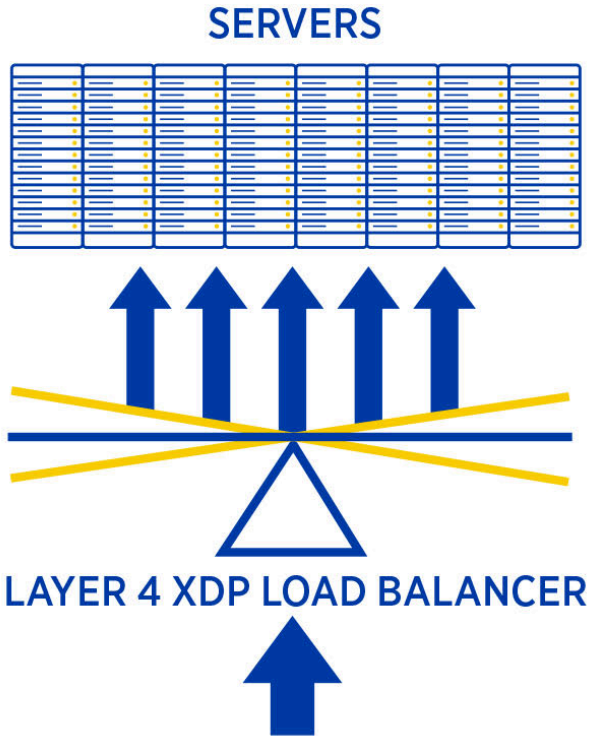
Map

Value	Action
0	XDP_ABORTED
1	XDP_DROP
2	XDP_PASS
3	XDP_TX

Demo 3 - Load Balancer

NETRONOME

Demo Source: https://github.com/Netronome/bpf-samples/tree/master/l4lb



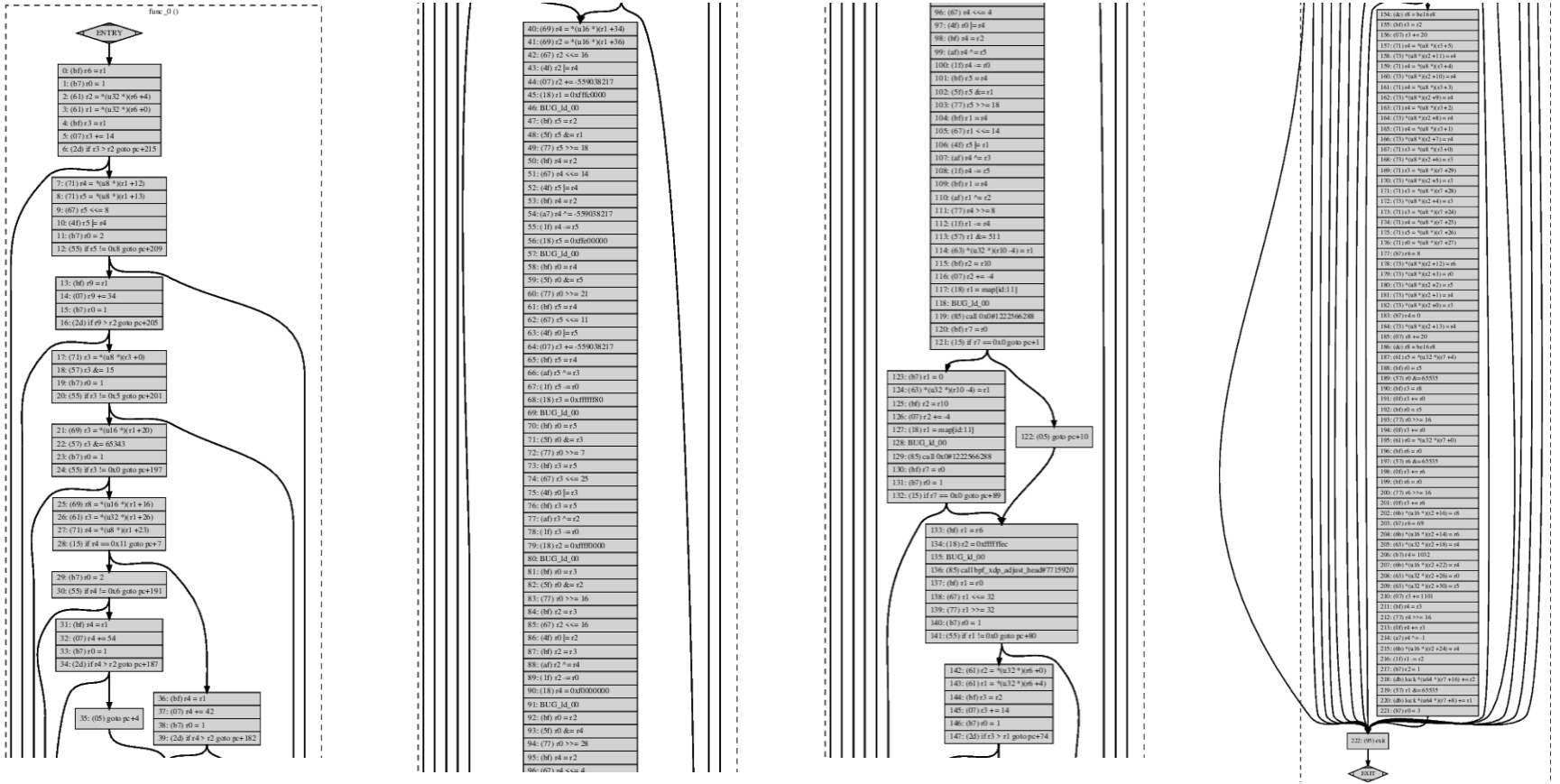
2.2.2.2	10.0.0.9
1.1.1.1	2.2.2.2
TCP	
1292	80

4 Tuple Hash

1.1.1.1	2.2.2.2
TCP	
1292	80

Hash Key	Server
0	10.0.0.1
1	10.0.0.6
2	10.0.0.9

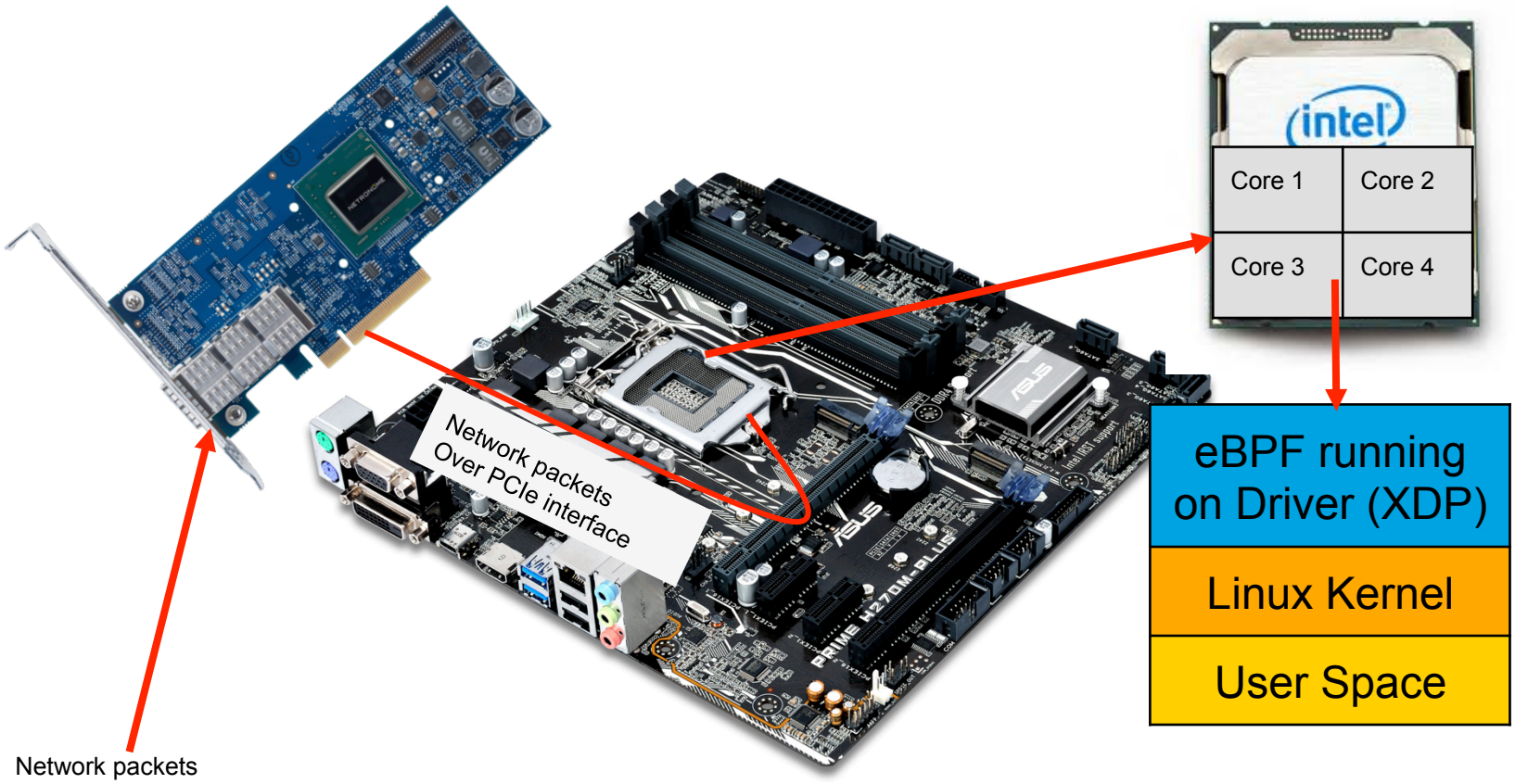
DAG Example - Load Balancer Demo



https://github.com/Netronome/bpf-samples/tree/master/l4lb

XDP Offload

NETRONOME



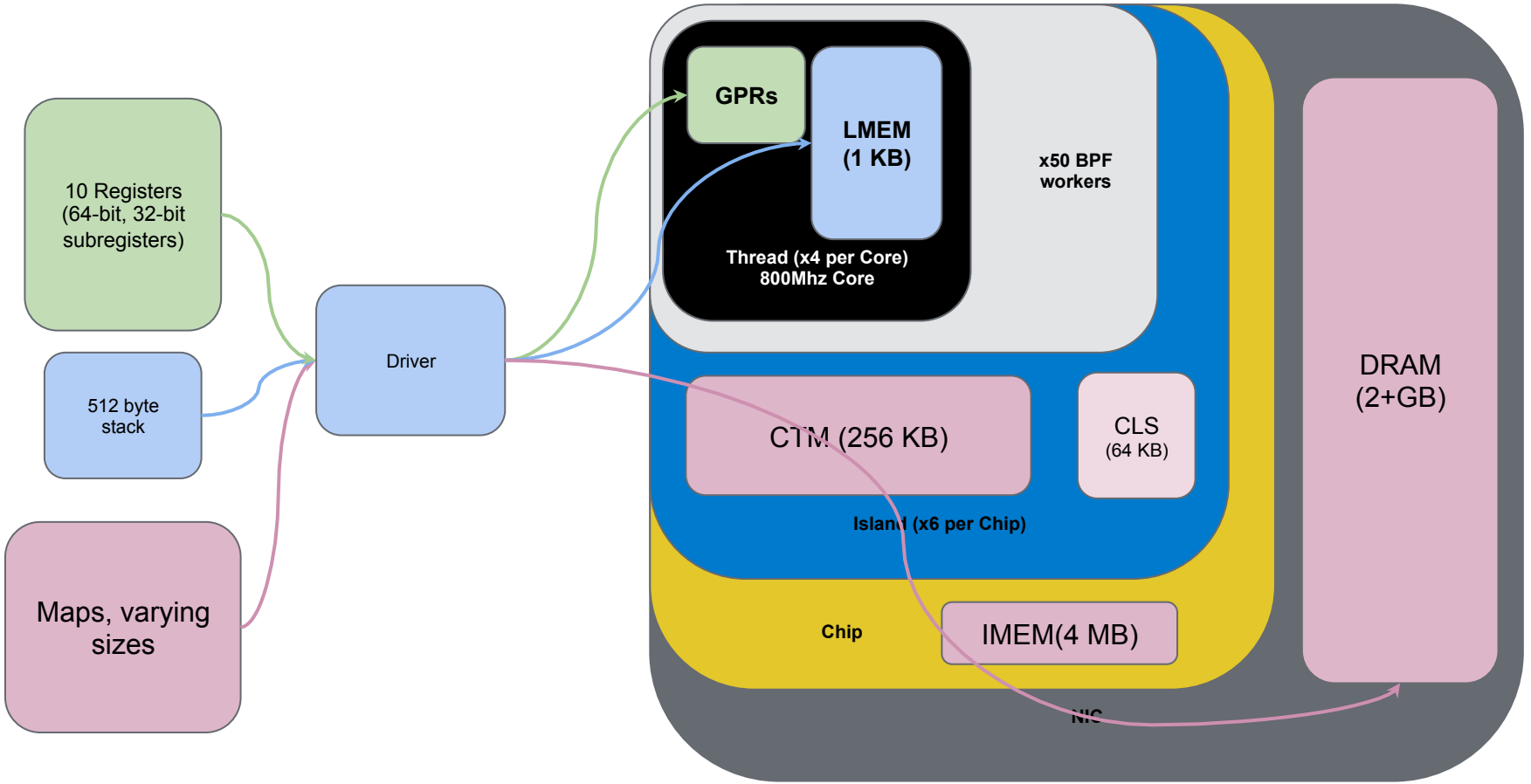
BPF for Host Datapath Acceleration

NETRONOME

- BPF VM provides a simple and well understood execution environment
- Most RISC cores should be able to execute JITed BPF code
- Kernel infrastructure improves, including verifier/analyzer, JIT compilers for all common host architectures and some common embedded architectures like ARM or x86
- Unlike higher level languages BPF is a intermediate representation (IR) which provides binary compatibility
- Advanced networking devices are capable of creating appropriate sandboxes
- Android APF targets smaller processors in mobile handsets for filtering wake ups from remote processors (most likely network interfaces) to improve battery life
- Linux kernel community is very active in extending performance and improving BPF feature set, with AF_XDP being a most recent example
- BPF is extensible through helpers and maps allowing us to make use of special HW features (when gain justifies the effort)

Kernel Offload - BPF Offload Memory Mapping

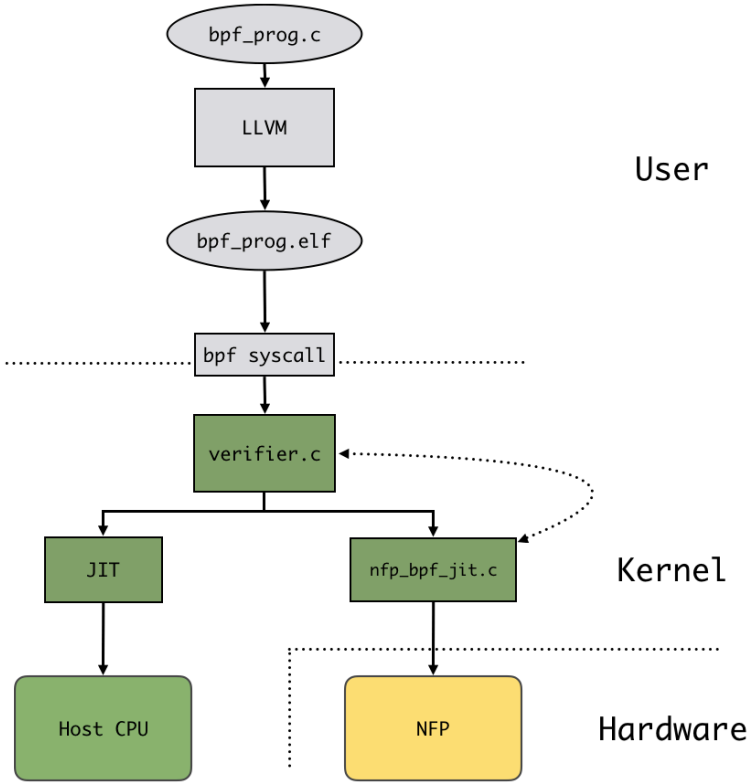
NETRONOME



Kernel Offload - Programming Model

NETRONOME

- LLVM compilation as normal
- iproute/tc/libbpf loads the program as normal but specifying “offload enable” flag
- maps are created on the device
- kernel directs the eBPF program to nfp/src/bpf/jit.c to converts to NFP machine code
- translation reuses the kernel verifier infrastructure for analysis
- **full ABI compatibility with the in-kernel BPF**



NFP JIT

NETRONOME

- LLVM optimizations can tune the code for BPF or even NFP BPF
- JIT steps:
  - preparation - build data structures
  - analysis - uses kernel verifier infrastructure
  - code generation
  - loading/relocation
- two pass translator:
  - convert memory accesses
  - inline helpers

Linux kernel: driver/net/ethernet/netronome/nfp/bpf/jit.c

GitHub: Netronome/nfp-drv-kmods/blob/master/src/bpf/jit.c

```
3030  static int jset_reg(struct nfp_prog *nfp_prog, struct nfp_insn_meta *meta)
3031  {
3032          return wrp_test_reg(nfp_prog, meta, ALU_OP_AND, BR_BNE);
3033  }
3034
3035  static int jne_reg(struct nfp_prog *nfp_prog, struct nfp_insn_meta *meta)
3036  {
3037          return wrp_test_reg(nfp_prog, meta, ALU_OP_XOR, BR_BNE);
3038  }
3039
3040  static int call(struct nfp_prog *nfp_prog, struct nfp_insn_meta *meta)
3041  {
3042          switch (meta->insn.imm) {
3043          case BPF_FUNC_xdp_adjust_head:
3044                  return adjust_head(nfp_prog, meta);
3045          case BPF_FUNC_map_lookup_elem:
3046          case BPF_FUNC_map_update_elem:
3047          case BPF_FUNC_map_delete_elem:
3048                  return map_call_stack_common(nfp_prog, meta);
3049          case BPF_FUNC_get_prandom_u32:
3050                  return nfp_get_prandom_u32(nfp_prog, meta);
3051          case BPF_FUNC_perf_event_output:
3052                  return nfp_perf_event_output(nfp_prog, meta);
3053          default:
3054                  WARN_ONCE(1, "verifier allowed unsupported function\n");
3055                  return -EOPNOTSUPP;
3056          }
3057  }
3058
3059  static int goto_out(struct nfp_prog *nfp_prog, struct nfp_insn_meta *meta)
3060  {
3061          emit_br_relo(nfp_prog, BR_UNC, BR_OFF_RELO, 0, RELO_BR_GO_OUT);
3062
3063          return 0;
3064  }
3065
3066  static const instr_cb_t instr_cb[256] = {
3067          [BPF_ALU64 | BPF_MOV | BPF_X] = mov_reg64,
3068          [BPF_ALU64 | BPF_MOV | BPF_K] = mov_imm64,
3069          [BPF_ALU64 | BPF_XOR | BPF_X] = xor_reg64,
3070          [BPF_ALU64 | BPF_XOR | BPF_K] = xor_imm64,
3071          [BPF_ALU64 | BPF_AND | BPF_X] = and_reg64,
3072          [BPF_ALU64 | BPF_AND | BPF_K] = and_imm64,
3073          [BPF_ALU64 | BPF_OR | BPF_X] =  or_reg64,
3074          [BPF_ALU64 | BPF_OR | BPF_K] =  or_imm64,
3075          [BPF_ALU64 | BPF_ADD | BPF_X] = add_reg64,
3076          [BPF_ALU64 | BPF_ADD | BPF_K] = add_imm64,
3077          [BPF_ALU64 | BPF_SUB | BPF_X] = sub_reg64,
3078          [BPF_ALU64 | BPF_SUB | BPF_K] = sub_imm64,
3079          [BPF_ALU64 | BPF_MUL | BPF_X] = mul_reg64,
3080          [BPF_ALU64 | BPF_MUL | BPF_K] = mul_imm64,
3081          [BPF_ALU64 | BPF_DIV | BPF_X] = div_reg64,
3082          [BPF_ALU64 | BPF_DIV | BPF_K] = div_imm64,
3083          [BPF_ALU64 | BPF_NEG] =         neg_reg64,
```

NFP JIT Example

NETRONOME

```
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
```

JITed into NFP Microcode

```
Bpftool prog dump jited id 1
   0:             .0  immed[gprB_6, 0x3fff]
   8:             .1  alu[gprB_6, gprB_6, AND, *l$index1]
  10:             .2  immed[gprA_0, 0x0], gpr_wrboth
  18:             .3  immed[gprA_1, 0x0], gpr_wrboth
  20:             .4  alu[gprA_4, gprB_6, +, *l$index1[2]], gpr_wrboth
  28:             .5  immed[gprA_5, 0x0], gpr_wrboth
  30:             .6  alu[gprA_2, --, B, *l$index1[2]], gpr_wrboth
  38:             .7  immed[gprA_3, 0x0], gpr_wrboth
  40:             .8  alu[gprA_6, --, B, gprB_2], gpr_wrboth
  48:             .9  alu[gprA_7, --, B, gprB_3], gpr_wrboth
  50:            .10  alu[gprA_6, gprA_6, +, 0xe], gpr_wrboth
  58:            .11  alu[gprA_7, gprA_7, +carry, 0x0], gpr_wrboth
  60:            .12  alu[--, gprA_4, -, gprB_6]
  68:            .13  alu[--, gprA_5, -carry, gprB_7]
  70:            .14  bcc[.33]
  78:            .15  immed[gprA_0, 0x1], gpr_wrboth
  80:            .16  immed[gprA_1, 0x0], gpr_wrboth
  88:            .17  mem[read32_swap, $xfer_0, gprA_2, 0xc, 1],
ctx_swap[sig1]
  90:            .18  ld_field_w_clr[gprA_4, 0001, $xfer_0], gpr_wrboth
  98:            .19  immed[gprA_5, 0x0], gpr_wrboth
  a0:            .20  alu[--, gprA_4, XOR, 0x22]
  a8:            .21  bne[.33]
  b0:            .22  alu[--, gprA_5, XOR, 0x0]
  b8:            .23  bne[.33]
  c0:            .24  ld_field_w_clr[gprA_2, 0001, $xfer_0, >>8],
gpr_wrboth
  c8:            .25  immed[gprA_3, 0x0], gpr_wrboth
  d0:            .26  immed[gprA_0, 0x2], gpr_wrboth
  d8:            .27  immed[gprA_1, 0x0], gpr_wrboth
                      …
```

Al … d support; no…

JIT Optimizations

NETRONOME

We can identify from assembly code certain sequences that can be replaced with fewer/faster NFP instructions, e.g.:

- memcpy(new_eth, old_eth, sizeof(*old_eth))
- Rotation
- ALU operation + register move
- bit operations
- compare and jump

32-bit subregister use; batching atomic operations; optimizing out helpers, e.g.:

- packet extend
- memory lookups

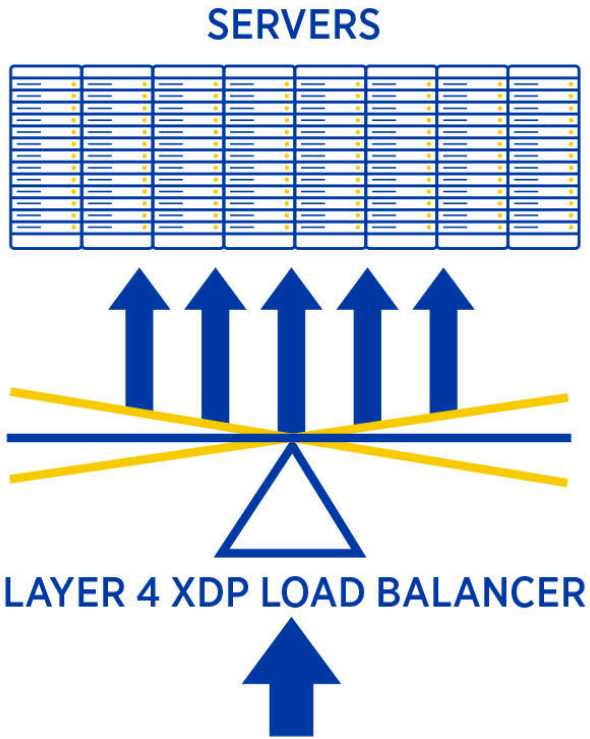
Creating read-only maps on the device

```
41:       71 14 21 00 00 00 00 00         r4 = *(u8 *)(r1 + 33)
42:       73 41 0d 00 00 00 00 00         *(u8 *)(r1 + 13) = r4
43:       71 14 20 00 00 00 00 00         r4 = *(u8 *)(r1 + 32)
44:       73 41 0c 00 00 00 00 00         *(u8 *)(r1 + 12) = r4
45:       71 14 1f 00 00 00 00 00         r4 = *(u8 *)(r1 + 31)
46:       73 41 0b 00 00 00 00 00         *(u8 *)(r1 + 11) = r4
47:       71 14 1e 00 00 00 00 00         r4 = *(u8 *)(r1 + 30)
48:       73 41 0a 00 00 00 00 00         *(u8 *)(r1 + 10) = r4
49:       71 14 1d 00 00 00 00 00         r4 = *(u8 *)(r1 + 29)
50:       73 41 09 00 00 00 00 00         *(u8 *)(r1 + 9) = r4
51:       71 14 1c 00 00 00 00 00         r4 = *(u8 *)(r1 + 28)
52:       73 41 08 00 00 00 00 00         *(u8 *)(r1 + 8) = r4
53:       71 14 1b 00 00 00 00 00         r4 = *(u8 *)(r1 + 27)
54:       73 41 07 00 00 00 00 00         *(u8 *)(r1 + 7) = r4
55:       71 14 1a 00 00 00 00 00         r4 = *(u8 *)(r1 + 26)
56:       73 41 06 00 00 00 00 00         *(u8 *)(r1 + 6) = r4
57:       71 14 19 00 00 00 00 00         r4 = *(u8 *)(r1 + 25)
58:       73 41 05 00 00 00 00 00         *(u8 *)(r1 + 5) = r4
59:       71 14 18 00 00 00 00 00         r4 = *(u8 *)(r1 + 24)
60:       73 41 04 00 00 00 00 00         *(u8 *)(r1 + 4) = r4
61:       71 14 17 00 00 00 00 00         r4 = *(u8 *)(r1 + 23)
62:       73 41 03 00 00 00 00 00         *(u8 *)(r1 + 3) = r4
63:       71 14 16 00 00 00 00 00         r4 = *(u8 *)(r1 + 22)
64:       73 41 02 00 00 00 00 00         *(u8 *)(r1 + 2) = r4
65:       71 14 15 00 00 00 00 00         r4 = *(u8 *)(r1 + 21)
66:       73 41 01 00 00 00 00 00         *(u8 *)(r1 + 1) = r4
67:       71 14 14 00 00 00 00 00         r4 = *(u8 *)(r1 + 20)
68:       73 41 00 00 00 00 00 00         *(u8 *)(r1 + 0) = r4
```

Demo 4 - Load Balancer on Offload

NETRONOME

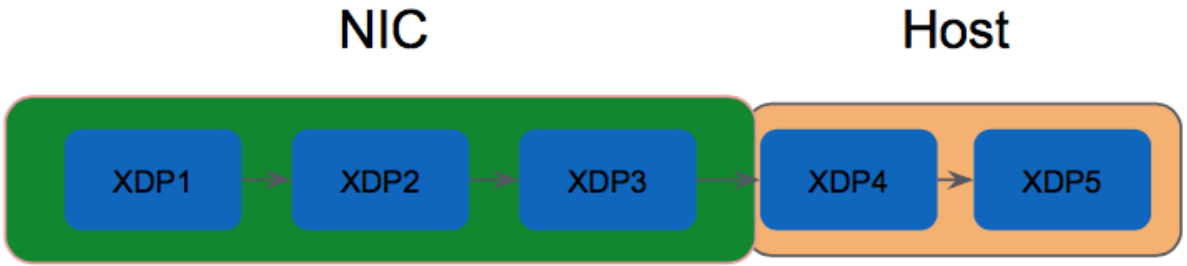
Demo Source: https://github.com/Netronome/bpf-samples/tree/master/l4lb



Kernel Offload - Multi-Stage Processing

NETRONOME

- 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



Redefining NIC Behavior

NETRONOME

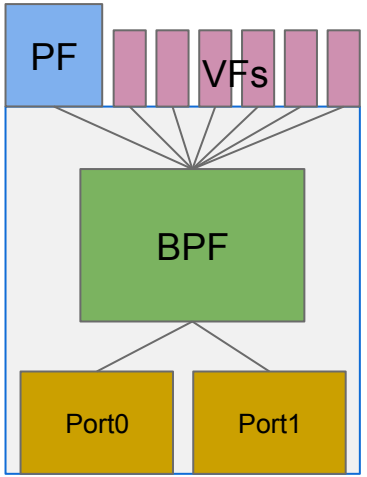
BPF offload allows users to change standard NIC features, e.g.:

- RSS
  - Users can create their own RSS schemes and parse arbitrary protocols
  - On standard NIC all packets go to queue 0 if protocols can't be parsed
  - More examples schemes in presentation about demos
- Flow affinity - similarly to RSS any flow affinity to RX queues can be defined
- SR-IOV forwarding (future)
  - With upcoming kernel extensions users will be able to define SR-IOV datapath in BPF
  - BPF-defined filtering and forwarding in HW
  - Any custom encapsulation/overlay supported

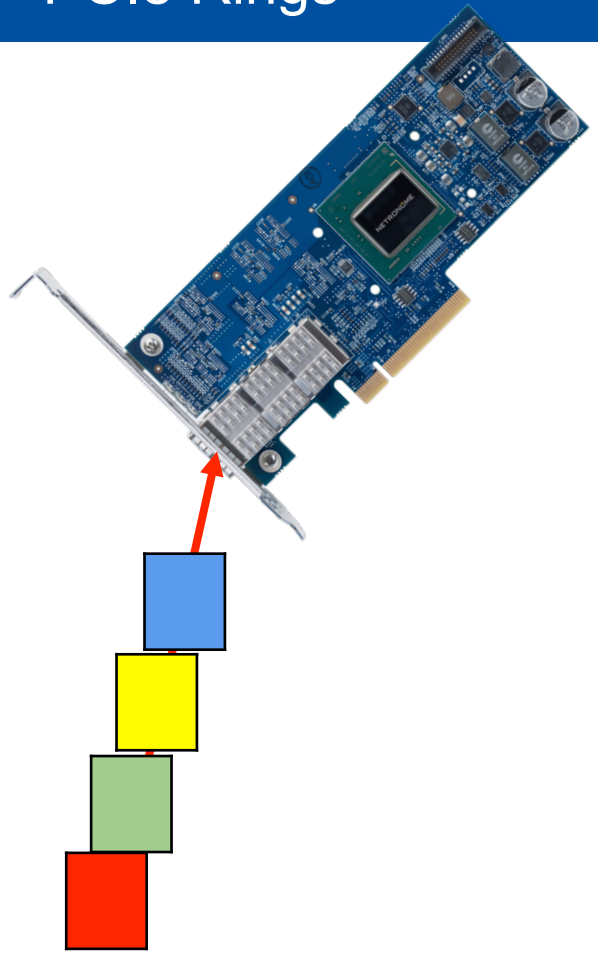
Switching with eBPF (incl. SR-IOV)

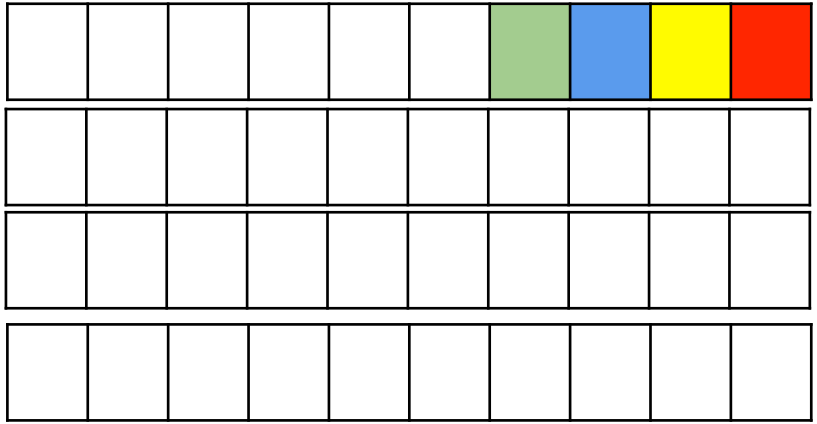
NETRONOME

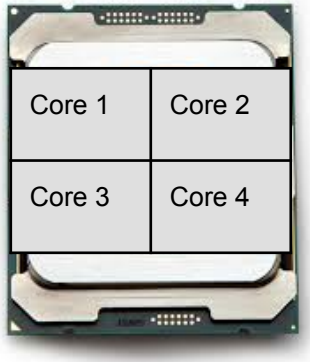
- full switchdev mode
  - Linux term for representing all ports as interfaces
- XDP ingress on all reprs (just link TC forwarding)
- XDP_REDIRECT support for forwarding decisions
- fallback path driver XDP? AF_XDP? up to users
- per-ASIC program and map sharing
- ingress device from xdp_rxq_info
- dealing with mcast/bcast requires a new BPF helper



PCIe Rings



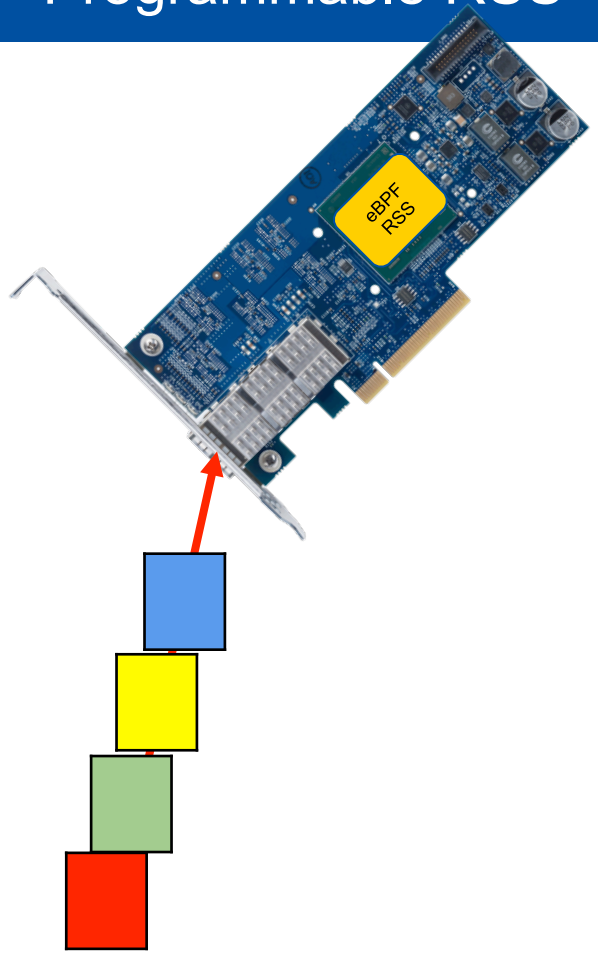


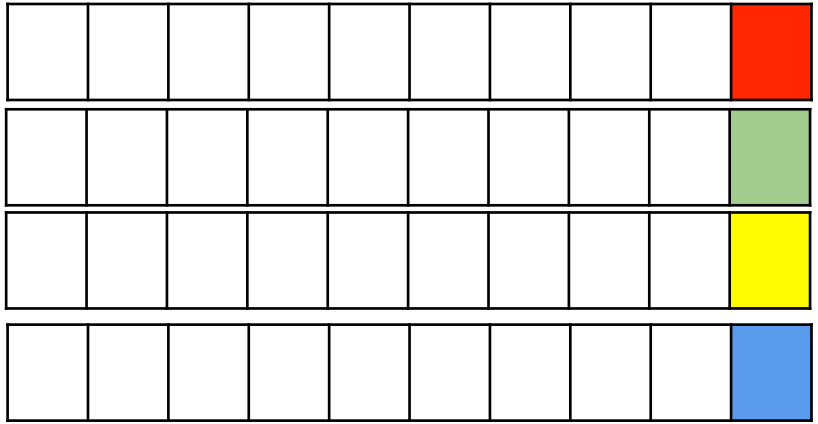


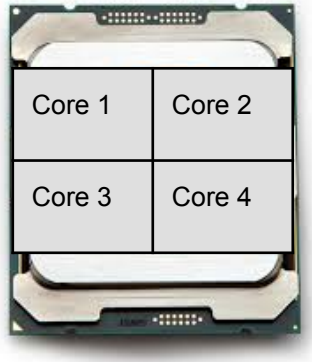
The queue is chosen using a hash on the header values, such as:

- IP Addresses
- UDP/TCP port numbers

Programmable RSS







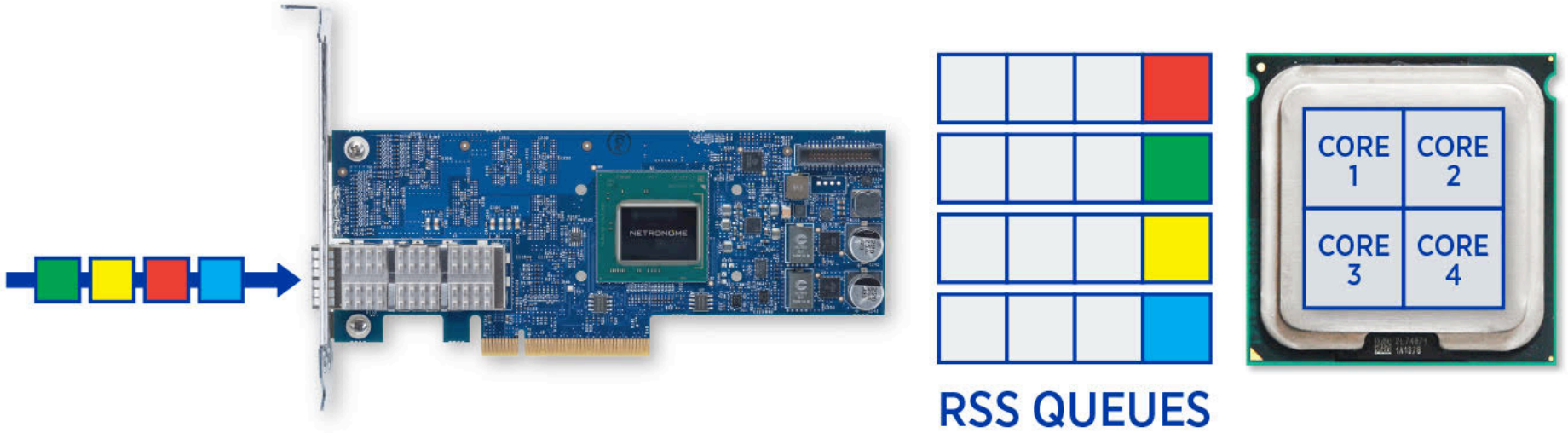
User programmable RSS

- Hash on payload headers
- Hash on inner IP headers

Demo 5 - Programmable RSS

NETRONOME

https://github.com/Netronome/bpf-samples/tree/master/programmable_rss



Offload Support

NETRONOME

Category	Functionality	Kernel 4.16	Kernel 4.17	Kernel 4.18	Near Future
eBPF offload program features	XDP_DROP				
	XDP_PASS				
	XDP_TX				
	XDP_ABORTED				
	Packet read access				
	Conditional statements				
	xdp_adjust_head()				
	bpf_get_prandom_u32()				
	perf_event_output()				
	RSS rx_queue_index selection				
	bpf_tail_call()				
	bpf_adjust_tail()				
eBPF offload map features	Hash maps				
	Array maps				
	bpf_map_lookup_elem()				
	bpf_map_delete_elem()				
	Atomic write (sync_fetch_and_add)				
eBPF offload performance optimizations	Localized packet cache				
	32-bit BPF support				

How to Participate with eBPF?

NETRONOME

Netronome Guides and Firmware

- https://help.netronome.com/support/solutions/folders/36000172266

Demo Applications

- https://github.com/Netronome/bpf-samples