NETRONOME

eBPF Offload Getting Started Guide

Netronome CX SmartNIC

Revision 1.0 − April 2018

Introduction

Netronome supports eBPF offload for XDP and cls_bpf on the Network Flow Processor (NFP). There are three components involved:

1. Agilio CX NIC
2. Linux Kernel
3. Compatible NFP Firmware

Agilio CX NIC

The Agilio CX NIC is a half-height, half-width NIC based on the NFP-4000. This is a 60-core processor with up to 8 cooperatively multithreaded threads per core (but eBPF programs are typically executed on 50 cores, each running 4 threads). The flow processing cores have a RISC instruction set that is optimized for networking. This instruction set is similar to eBPF bytecode, ensuring the offload is a viable proposition.

Kernel support

Netronome is currently upstreaming changes to the Linux kernel. eBPF hardware offload support appeared in kernel 4.9, but feature additions continue to be made. This document focuses on the latest stable version at this date, kernel v4.16, which is also the first kernel to have map offload support.

The upstreamed kernel driver allows for the translation of the kernel eBPF program into microcode which can be transferred onto our network cards via the NFP eBPF Just-in-Time (JIT) compiler. This allows for users to offload programs without requiring any microcode knowledge or understanding of our architecture by using eBPF.

NFP Firmware

The network card requires an eBPF compatible firmware to enable the functionality. This firmware is loaded from /lib/firmware/netronome/nic_xxx…..nffw. The firmware is available in package form from our public support site and will be added to the Linux Kernel FW repo in the near future.

Kernel version support

Category	Functionality	Kernel 4.16	Kernel 4.17	Near Future*
eBPF offload program features	XDP_DROP	■	■	■
	XDP_PASS	■	■	■
	XDP_TX	■	■	■
	XDP_ABORTED	■	■	■
	Packet read access	■	■	■
	Conditional statements	■	■	■
	xdp_adjust_head()	■	■	■
	bpf_tail_call()			■
	bpf_get_random()		■	■
	perf_event_output()			■
eBPF offload map features	Offload ownership for maps	■	■	■
	Hash maps	■	■	■
	Array maps	■	■	■
	bpf_map_lookup_elem()	■	■	■
	bpf_map_delete_elem()		■	■
	Atomic write actions (sync_fetch_and_add)		■	■
eBPF offload performance optimizations	32 bit BPF support		■	■
	Localized packet cache		■	■
	Localized maps			■

* timelines are subject to change

Environment Setup

Kernel

Kernel 4.17 is highly recommended for offloading eBPF / XDP to the NFP. To build the kernel from source, follow the steps below.

1. Download required libraries.

```
# apt-get install make gcc libelf-dev bc build-essential binutils-dev ncurses-dev
libssl-dev util-linux pkg-config elfutils libreadline-dev
```

2. Clone the kernel repository.

```
$ git clone https://github.com/torvalds/linux.git ~/kernel
```

3. Setup the kernel build configuration.

```
$ cp /boot/config-`uname -r` ~/kernel/.config
$ cd ~/kernel/
$ make olddefconfig
```

4. Ensure that NFP and BPF are enabled within the kernel .config file.

```
CONFIG_NFP=m
CONFIG_NFP_DEBUG=y
CONFIG_NET_DEVLINK=y
CONFIG_BPF=y
CONFIG_BPF_SYSCALL=y
```

5. Compile the kernel and modules.

```
$ make -j (number of cores)
```

6. Install the kernel onto the system.

```
# make modules_install
# make install
```

7. Reboot the system.

8. Check the kernel version to ensure it has booted into the new kernel.

```
$ uname -a
```

Firmware

Download the agilio-bpf firmware files for the relevant distribution. Install the files using the following command.

1. For Debian/Ubuntu:

```
# dpkg -i agilio-bpf-firmware-XXXX.deb
```

2. For RedHat/Centos:

```
# rpm -i agilio-bpf-firmware-XXXX.rpm
```

3. Update the NFP driver symbolic links to point to the eBPF firmware.

```
$ cd /lib/firmware/netronome
# ln -s agilio-bpf/* .
```

Driver

The nfp driver required for eBPF offload is shipped with the kernel and should have been automatically installed on your system when installing the new kernel. When it is inserted into the kernel, the driver searches for a compatible firmware to load to the card. Follow those steps to make sure the newly firmware is loaded:

1. Remove and reload the driver.

```
# modprobe -r nfp
# modprobe nfp
```

2. Check `dmesg` logs that eBPF capability has been enabled within the driver.

```
$ dmesg
    [...]
    nfp 0000:81:00.0: nfp: netronome/nic_AMDA0081-0001_1x40.nffw: found,
    nfp 0000:81:00.0: Soft-reset, loading FW image
    nfp 0000:81:00.0: Finished loading FW image
    nfp 0000:81:00.0 eth0: CAP: 0x78140233 PROMISC RXCSUM TXCSUM GATHER TSO2 RSS2
                               AUTOMASK IRQMOD RXCSUM_COMPLETE BPF
    nfp 0000:04:00.0 ens4: renamed from eth1
```

3. Check `ip link` output for the interface status.

```
$ ip link
    18: ens4: mtu 1500 qdisc noop state DOWN mode DEFAULT group default qlen 1000
        link/ether 00:15:4d:12:1d:79 brd ff:ff:ff:ff:ff:ff
```

4. `ethtool` can also be used to check that the firmware has eBPF offload capability.

```
$ ethtool -i $ETHNAME
    driver: nfp
    version: 45443c7 (o-o-t)
    firmware-version: 0.0.3.5 0.22 bpf_9c3a83 ebpf
```

Setting up rings and affinities

We recommend running the following commands for each interface to provide it with sufficient resources for when eBPF runs in driver mode. In this example, we have a server with 8 cores, therefore we are allocating 8 rings. The IRQ affinity script can be obtained from our public driver repository: https://github.com/Netronome/nfp-drv-kmods/blob/master/tools/set_irq_affinity.sh

```
# ifconfig $ETHNAME 10.0.0.4 up mtu 1500
# numactl -m 0 -N 0 ethtool -L $ETHNAME rx 0 tx 0 combined 8
# numactl -m 0 -N 0 ethtool -G $ETHNAME rx 512 tx 512
# ~/nfp-drv-kmods/tools/set_irq_affinity.sh $ETHNAME
```

iproute2 utilities

A version after January 2018 is required for NFP offload. It is recommended to download the latest iproute2 package using the iproute2-next public repository.

1. Clone the sources.

```
$ git clone https://git.kernel.org/pub/scm/network/iproute2/iproute2-next.git
```

2. Install required dependencies.

```
# apt-get install libelf-dev libmnl-dev bison flex
```

3. Compile iproute2 tools and check for libelf support.

```
$ make
[...]
ELF support: yes
[...]
# make install
```

4. Check the installed `ip` version.

```
$ ip -V
```

```
   ip utility, iproute2-ss180129
```

Clang Compiler

Clang 4.0 is required to carry out simple eBPF compilation. However we recommend clang 6.0 is used to provide optimized compilation.
Ubuntu 16.04 uses `clang-4.0` in its APT repository. In this case, the addition of the LLVM repository is required to get access to `clang-6.0`.

1. Go to https://apt.llvm.org/ and add the relevant repository to your OS.
   For example, for Ubuntu 16.04 (Xenial) add the following to /etc/apt/source.list:

```
deb http://apt.llvm.org/xenial/ llvm-toolchain-xenial-6.0 main
deb-src http://apt.llvm.org/xenial/ llvm-toolchain-xenial-6.0 main
```

2. Retrieve the key for the repository.

```
# wget -O - https://apt.llvm.org/llvm-snapshot.gpg.key|sudo apt-key add -
    ## Fingerprint should be: 6084 F3CF 814B 57C1 CF12 EFD5 15CF 4D18 AF4F 7421
```

3. Install `clang-6.0`.

```
# apt-get update
# apt-get install clang-6.0
```

4. Update system clang to point to the now installed `clang-6.0`.

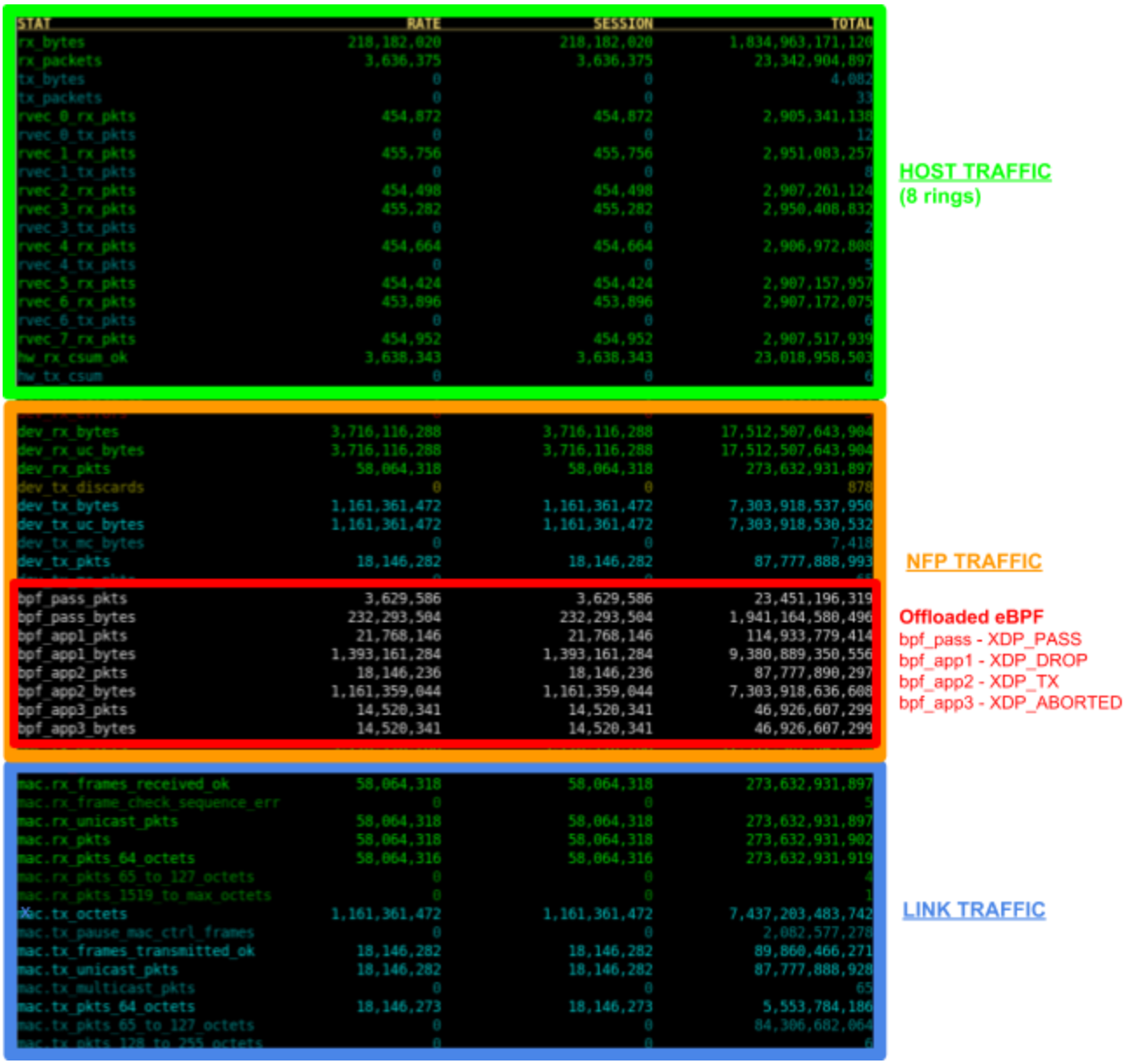
```
# update-alternatives --install /usr/bin/clang     clang     /usr/bin/clang-6.0 100
# update-alternatives --install /usr/bin/clang++   clang++   /usr/bin/clang++-6.0 100
# update-alternatives --install /usr/bin/llc       llc       /usr/bin/llc-6.0 100
# update-alternatives --install /usr/bin/llvm-mc   llvm-mc   /usr/bin/llvm-mc-6.0 50
```

Ubuntu 18.04 offers `clang-6.0`, so there is no need to install the LLVM repository on that system. Other distributions may use different package managers and require different commands, please consult the relevant instructions available at https://apt.llvm.org if you need to update tol `clang-6.0` or higher.

Stat Watch

`stat_watch.py` is a tool we provide within our public GitHub driver repository (https://github.com/Netronome/nfp-drv-kmods/blob/master/tools/stat_watch.py). It displays `ethtool` measurements values in table form. It can be accessed using the command below:

```
$  ~/nfp-drv-kmods/tools/stat_watch.py $ETHNAME -c
```



Offloading a basic eBPF program

1. Create the following program and save it as drop.c.

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

int xdp_prog1(struct xdp_md *ctx __attribute__((unused))) {
       return XDP_DROP;
}
```

2. Compile the program using `clang`.

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

3. Offload the program using `ip link` (change `$ETHNAME` to the relevant interface).

```
# ip -force link set dev $ETHNAME xdpoffload obj drop.o sec .text
```

4. Check that the program is offloaded using `ip link`.

```
$ ip link show dev $ETHNAME
    18: ens4: <BROADCAST,MULTICAST> mtu 1500 xdpoffload qdisc noop state UP mode
               DEFAULT group default qlen 1000
               link/ether 00:15:4d:12:1d:79 brd ff:ff:ff:ff:ff:ff
               prog/xdp id 35 tag 57cd311f2e27366b jited
```

5. Send traffic to the interface and check `stat_watch.py`. All packets should be dropped, represented in stat watch by field `bpf_app1`.

STAT	RATE	SESSION	TOTAL
tx_bytes	0	2,378	2,378
tx_packets	0	19	19
rvec_3_tx_pkts	0	8	8
rvec_10_tx_pkts	0	11	11
hw_tx_csum	0	6	6
dev_rx_errors	0	1	1
dev_rx_bytes	3,822,406,208	198,012,270,976	198,012,270,976
dev_rx_uc_bytes	3,822,406,208	198,012,270,976	198,012,270,976
dev_rx_pkts	59,725,096	3,093,941,756	3,093,941,756
dev_tx_bytes	0	2,454	2,454
dev_tx_mc_bytes	0	2,454	2,454
dev_tx_pkts	0	19	19
dev_tx_mc_pkts	0	19	19
bpf_app1_pkts	59,725,098	3,093,943,411	3,093,943,411
bpf_app1_bytes	3,822,406,152	198,012,379,144	198,012,379,144

6. Now remove the offloaded program from the interface.

```
# ip -force link set dev $ETHNAME xdpoffload off
```

The above steps can be repeated to perform `XDP_PASS` (bpf_pass), `XDP_TX` (bpf_app2), `XDP_ABORTED` (bpf_app3). Note that the app codes are related to those used in cls_bpf for historical reasons.

Advanced programming

Available helpers

The list of eBPF helper functions that can be called from within an eBPF program and are currently implemented by the NFP is the following:

```
void *bpf_map_lookup_elem(struct bpf_map *map, void *key)
      Perform a lookup in map for an entry associated to key.
      Return: Map value associated to key, or NULL if no entry was found.

int bpf_map_delete_elem(struct bpf_map *map, void *key)
      Delete entry with key from map.
      Return: 0 on success, or a negative error in case of failure.

u32 bpf_get_prandom_u32(void)
      Return a random 32-bit unsigned value.

int bpf_xdp_adjust_head(struct xdp_buff *xdp_md, int delta)
      Adjust (move) xdp_md->data by delta bytes. Note that it is possible to use
      a negative value for delta. This helper can be used to prepare the packet
      for pushing or popping headers.
      A call to this helper is susceptible to change data from the packet.
      Therefore, at load time, all checks on pointers previously done by the
      verifier are invalidated and must be performed again.
      Return: 0 on success, or a negative error in case of failure.
```

Atomic writes

As of this writing, updating maps from the offloaded eBPF program is not supported (maps can be updated from user space, for example with bpftool, see related section). However, the NFP supports basic atomic write operations (fetch-and-add). Here is an example:

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

struct bpf_map_def SEC("maps") map_cnt = {
        .type = BPF_MAP_TYPE_ARRAY,
        .key_size = sizeof(__u32),
        .value_size = sizeof(__u64),
        .max_entries = 1024,
};

SEC("xdp")
int xdp_prog1()
{
        __u32 key = 0;
        __u32 *count;

        count = bpf_map_lookup_elem(&map_cnt, &key);
        if (!count)
                return XDP_DROP;
        __sync_fetch_and_add(count, 1);
        return XDP_DROP;
}
char _license[] SEC("license") = "GPL";
```

User space control of offloaded eBPF

Access to eBPF objects

User space programs can interact with the offloaded program in the same way as normal eBPF programs. The kernel will try and offload the program if a non-null `ifindex` is supplied to the `bpf()` system call for loading the program.

Maps can be accessed from the kernel using user space eBPF map lookup/update commands (technically: the `bpf()` system call). An easy way to manage map entries consists in using bpftool that relies on such commands.

bpftool

bpftool is a user space utility used for introspection and management of eBPF objects (maps and programs). It is not packaged on distributions as of this writing, so it is necessary to compile it from the sources, which are located within the Linux kernel tree. Follow the steps below to install it on your system.

1. Install the required dependencies. Note that you may have installed `binutils-dev` and `libelf-dev` already before installing the kernel and iproute2, respectively. Package `python-docutils` is only required for building the documentation.

```
# apt install binutils-dev libelf-dev python-docutils
```

2. Compile the program and the documentation.

```
$ cd tools/bpf/bpftool
$ make
$ make doc
```

3. Install them on the system.

```
# make install doc-install
```

The documentation is installed as manual pages that you can access with the `man` utility:

```
$ man bpftool
$ man bpftool-prog
$ man bpftool-map
```

bpftool can be used to gather information about eBPF programs and maps. For example you can list loaded programs:

```
# bpftool prog show
1337: sched_cls  name cls_entry  tag e202124da7c84e89
      loaded_at Mar 08/19:53  uid 0
      xlated 304B  not jited  memlock 4096B
```

And you could dump the instructions (JIT-compiled or not) for this program:

```
# bpftool prog dump xlated id 1337
   0: (71) r6 = *(u8 *)(r1 +142)
   1: (54) (u32) r6 &= (u32) 1
   2: (15) if r6 == 0x0 goto pc+7
   3: (bf) r6 = r1
   [...]
  37: (95) exit
```

While unavailable as of this writing, dumping the code compiled for NFP when the program is offloaded is being pushed upstream and will soon be available.

Maps can be listed and dumped too:

```
# bpftool map
1234: array  name ch_rings  flags 0x0
      key 4B  value 4B  max_entries 7860  memlock 65536B

# bpftool map dump id 1234
key: 00 00 00 00  value: 00 00 00 00
key: 01 00 00 00  value: 00 00 00 00
key: 02 00 00 00  value: 00 00 00 00
key: 03 00 00 00  value: 00 00 00 00
[...]
Found 7860 elements
```

It is also possible to execute some management operations, including (but not limited to) loading programs, performing lookups or updates of map values. Here is an example for the latter:

```
# bpftool map update id 1234 key 0x01 0x00 0x00 0x00
```

Debugging eBPF

This section is not exactly about eBPF offload, but provides some hints about how to debug eBPF programs, and also apply when the program is run on the SmartNIC.

bpftool, of course, can be used for introspection and debug (for example to dump the code of the program, or the contents of a given map). See the related section above.

LLVM

llvm-objdump

LLVM, and the front-end `clang`, are of course extremely useful to compile programs from C to eBPF bytecode. However, LLVM has also a number of other tools that can help with debugging. For instance, `llvm-objdump` (version 4.0 or higher) can be used to dump the compiled bytecode in a human-readable fashion, before the user tries to inject it into the kernel.

```
$ llvm-objdump-4.0 -S sample_ret0.o

sample_ret0.o:  file format ELF64-BPF

Disassembly of section .text:
func:
; {
       0:    b7 00 00 00 00 00 00 00       r0 = 0
; return 0;
       1:    95 00 00 00 00 00 00 00       exit
```

Flag `-g` must be passed to `clang` when compiling the program to get information about the C source code.

llvm-mc

With `llvm-mc`, LLVM version 6.0 and higher also provides an eBPF assembler. One can compile step by step: first from C to an eBPF-assembly representation and then to bytecode. This is particularly useful to test specific sequences of instructions, since it is not necessary to manually write the full program as hexadeximal instructions. Here is an example: let's compile a program that just returns 0 from C to eBPF assembly with `clang`.

```
$ clang -target bpf -S -o sample_ret0.S sample_ret0.c
$ cat sample_ret0.S
      .text
      .globl func             # -- Begin function func
      .p2align          3
      func:                   # @func
      # %bb.0:
      r0 = 0
      exit

                              # -- End function
```

The language used in this eBPF assembly is the same as the verifier output (note: there is no official human-readable eBPF assembly syntax, the form used by other tools may differ).

Let's edit the code:

```
$ sed -i 's/r0 = 0/r0 = -1/' sample_ret0.S
```

Now we can compile it with `llvm-mc` to produce the ELF object file:

```
$ llvm-mc -triple bpf -filetype=obj -o sample_ret.o sample_ret0.S
$ llvm-objdump-6.0 -d sample_ret0.o

sample_ret0.o: file format ELF64-BPF

Disassembly of section .text:
func:
      0: b7 00 00 00 ff ff ff ff    r0 = -1
      1: 95 00 00 00 00 00 00 00    exit
```

`log_level` flag for program load

When loading programs, the `bpf()` system call accepts a `log_level` attribute field which is used to set the level for debug. It can have the following values:

- 0: No debug output.
- 1: Debug information from the verifier (all instructions).
- 2: More information: add all register states after each instruction.

For example, here is the output for a program loaded with `log_level` set to 2.

```
0: R1=ctx R10=fp
0: (b7) r3 = 2
1: R1=ctx R3=imm2,min_value=2,max_value=2,min_align=2 R10=fp
1: (b7) r3 = 4
2: R1=ctx R3=imm4,min_value=4,max_value=4,min_align=4 R10=fp
2: (b7) r3 = 8
3: R1=ctx R3=imm8,min_value=8,max_value=8,min_align=8 R10=fp
3: (b7) r3 = 16
4: R1=ctx R3=imm16,min_value=16,max_value=16,min_align=16 R10=fp
4: (b7) r3 = 32
5: R1=ctx R3=imm32,min_value=32,max_value=32,min_align=32 R10=fp
5: (b7) r0 = 0
6: R0=imm0,min_value=0,max_value=0,min_align=2147483648 R1=ctx \
      R3=imm32,min_value=32,max_value=32,min_align=32 R10=fp
6: (95) exit
```

Not all tools propose an option to change this value. Currently, for passing it with `tc` or `ip`, patching iproute2 code is required. The following patch could be used to do so.

```
diff --git a/lib/bpf.c b/lib/bpf.c
index 2db151e4dd3c..1fd7daaba1e1 100644
--- a/lib/bpf.c
+++ b/lib/bpf.c
@@ -1082,7 +1082,7 @@ static int bpf_prog_load_dev(enum bpf_prog_type type,
      if (size_log > 0) {
      attr.log_buf = bpf_ptr_to_u64(log);
      attr.log_size = size_log;
-     attr.log_level = 1;
+     attr.log_level = 2;
      }

      return bpf(BPF_PROG_LOAD, &attr, sizeof(attr));
```

Further Reading

NFP Architecture

Open-NFP Classroom
https://open-nfp.org/the-classroom/

The Joy of Micro-C: This document contains information about the NFP architecture
https://open-nfp.org/m/documents/the-joy-of-micro-c_fcjSfra.pdf

eBPF Offload

Netdev 2.2 talk (Nov 2017) - *Comprehensive XDP Offload: Handling the Edge Cases*
https://www.youtube.com/watch?v=3qEbPSqq-QI

Transparent eBPF Offload: eBPF hardware offload advice
https://www.youtube.com/watch?v=W2v7zgUGp8A

eBPF and XDP

Kernel documentation
https://www.kernel.org/doc/Documentation/networking/filter.txt

Summary of eBPF instructions syntax and opcodes
https://github.com/iovisor/bpf-docs/blob/master/eBPF.md

Cilium BPF and XDP documentation
http://docs.cilium.io/en/latest/bpf/

BPF design Q & A, from kernel documentation
https://git.kernel.org/pub/scm/linux/kernel/git/davem/net-next.git/tree/Documentation/bpf/bpf_design_QA.txt

Manual pages for bpf() and TC with BPF filters

- http://man7.org/linux/man-pages/man2/bpf.2.html

- http://man7.org/linux/man-pages/man8/tc-bpf.8.html

David Miller's emails on xdp-newbies mailing list

- https://www.spinics.net/lists/xdp-newbies/msg00179.html *bpf.h and you...*
- https://www.spinics.net/lists/xdp-newbies/msg00181.html *Contextually speaking...*
- https://www.spinics.net/lists/xdp-newbies/msg00185.html *BPF Verifier Overview*

Kernel versions required for each BPF feature
https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md

BPF-related compilation of resources

https://qmonnet.github.io/whirl-offload/2016/09/01/dive-into-bpf/

Contact us

Netronome Systems, Inc.
2903 Bunker Hill Lane, Suite 150 Santa Clara, CA 95054
Tel: 408.496.0022 | Fax: 408.586.0002
www.netronome.com