NETRONOME

eBPF Offload Getting Started Guide

Netronome CX SmartNIC

Revision 1.2 − August 2018
Kernel 4.18

Introduction

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

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

Agilio CX SmartNIC

The Agilio CX SmartNIC 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 kernels available after v4.16 which provides 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 SmartNICs support site (https://help.netronome.com/) and will be added to the Linux Kernel firmware repository in the near future.

Kernel version support*

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_adjust_tail()				
	Partial offload				
eBPF offload map features	Offload ownership for maps				
	Hash maps				
	Array maps				
	bpf_map_lookup_elem()				
	bpf_map_update_elem()				
	bpf_map_delete_elem()				
	Atomic write (sync_fetch_and_add)				
	Map sharing between ports				
eBPF offload performance optimizations	Localized packet cache				
	32 bit BPF support				
	Localized maps				

* Timelines are subject to change

Environment Setup

We recommend using Ubuntu 18.04 or Fedora 28, due to these distributions having the latest packages available. Fedora 28 in particular is recommended, as a fresh install with the latest repository updates, will give the recommended kernel and iproute2 version.
Other distributions can be used but may require the necessary tools to be compiled from source. Relevant instructions for this are included in the Appendix.

Kernel

Kernel 4.17 or higher is highly recommended for offloading eBPF / XDP to the NFP. The current kernel version can be checked using the following command.

```
$ uname -r
4.18.1-1.vanilla.knurd.1.fc28.x86_64
```

Fedora 28

To update the Fedora 28 kernel to the latest stable kernel, run the following commands.

```
# curl -s https://repos.fedorapeople.org/repos/thl/kernel-vanilla.repo | sudo tee
/etc/yum.repos.d/kernel-vanilla.repo
# yum install kernel-devel
# dnf --enablerepo=kernel-vanilla-stable update
# reboot
```

If required, the latest pre-release kernel can be obtained from the follow repository.

```
# dnf --enablerepo=kernel-vanilla-mainline-wo-mergew update
# reboot
```

Ubuntu 18.04

Obtain the latest kernel from the official repository at http://kernel.ubuntu.com/~kernel-ppa/mainline. The following commands are for kernel 4.18.

```
$ wget
http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.18/linux-headers-4.18.0-041800_4.18.0-04
1800.201808122131_all.deb

$ wget
http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.18/linux-headers-4.18.0-041800-generic_4
.18.0-041800.201808122131_amd64.deb

$ wget
http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.18/linux-image-unsigned-4.18.0-041800-ge
neric_4.18.0-041800.201808122131_amd64.deb

$ wget
http://kernel.ubuntu.com/~kernel-ppa/mainline/v4.18/linux-modules-4.18.0-041800-generic_4
.18.0-041800.201808122131_amd64.deb
```

Then Install the packages.

```
# dpkg -i linux-headers-4.18.0-041800_4.18.0-041800.201808122131_all.deb
# dpkg -i linux-headers-4.18.0-041800-generic_4.18.0-041800.201808122131_amd64.deb
# dpkg -i linux-modules-4.18.0-041800-generic_4.18.0-041800.201808122131_amd64.deb
# dpkg -i linux-image-unsigned-4.18.0-041800-generic_4.18.0-041800.201808122131_amd64.deb
# reboot
```

Other Distributions

To build the kernel from source, follow the steps provided in the Appendix.

Firmware

Download the agilio-bpf firmware files for the relevant distribution from the "*Agilio eBPF Software*" knowledge base section of our SmartNICs support website https://help.netronome.com.

1. Install the files using the following command.

    - For Debian / Ubuntu:

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

    - For RedHat / Fedora / Centos:

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

2. 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 and ensure the interface state is UP.

```
$ ip link
    18: ens4: mtu 1500 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
```

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

```
$ ethtool -i $ETHNAME
    driver: nfp
    version: 4.17.1-250
    firmware-version: 0.0.3.5 0.22 bpf-2.0.6.121 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, at 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
```

Note: The maximum number of allowed rings for eBPF on driver mode is 31 combined per card. This allows for 31 rings on a single port card, and 15 queues per interface for dual port cards. This limitation does not apply to eBPF on offload.

iproute2 utilities

Iproute2 tagged newer than v4.16 (ss180402) is required for NFP offload.
Check the installed `ip` version to ensure that the version is newer than 2018-04. If not, follow the installation instructions below.

```
$ ip -V
   ip utility, iproute2-ss180402
```

Fedora 28

1. Install iproute2 from the updates-testing repository.

```
# dnf --enablerepo=updates-testing --best install iproute
```

Ubuntu 18.04 and other distributions

Currently there is no iproute2 binary available for Ubuntu, so compilation is required.

1. Clone the sources from the development repository.

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

2. Install required dependencies.

```
# apt-get install elfutils libelf-dev libmnl-dev bison flex pkg-config
```

3. Compile iproute2 tools and check for libelf and libmnl support.

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

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 18.04 and Fedora 28 offers `clang-6.0` in their upstream repository, so can be obtained using the inbuilt package manager.

To check the installed `clang` version, run the following command.

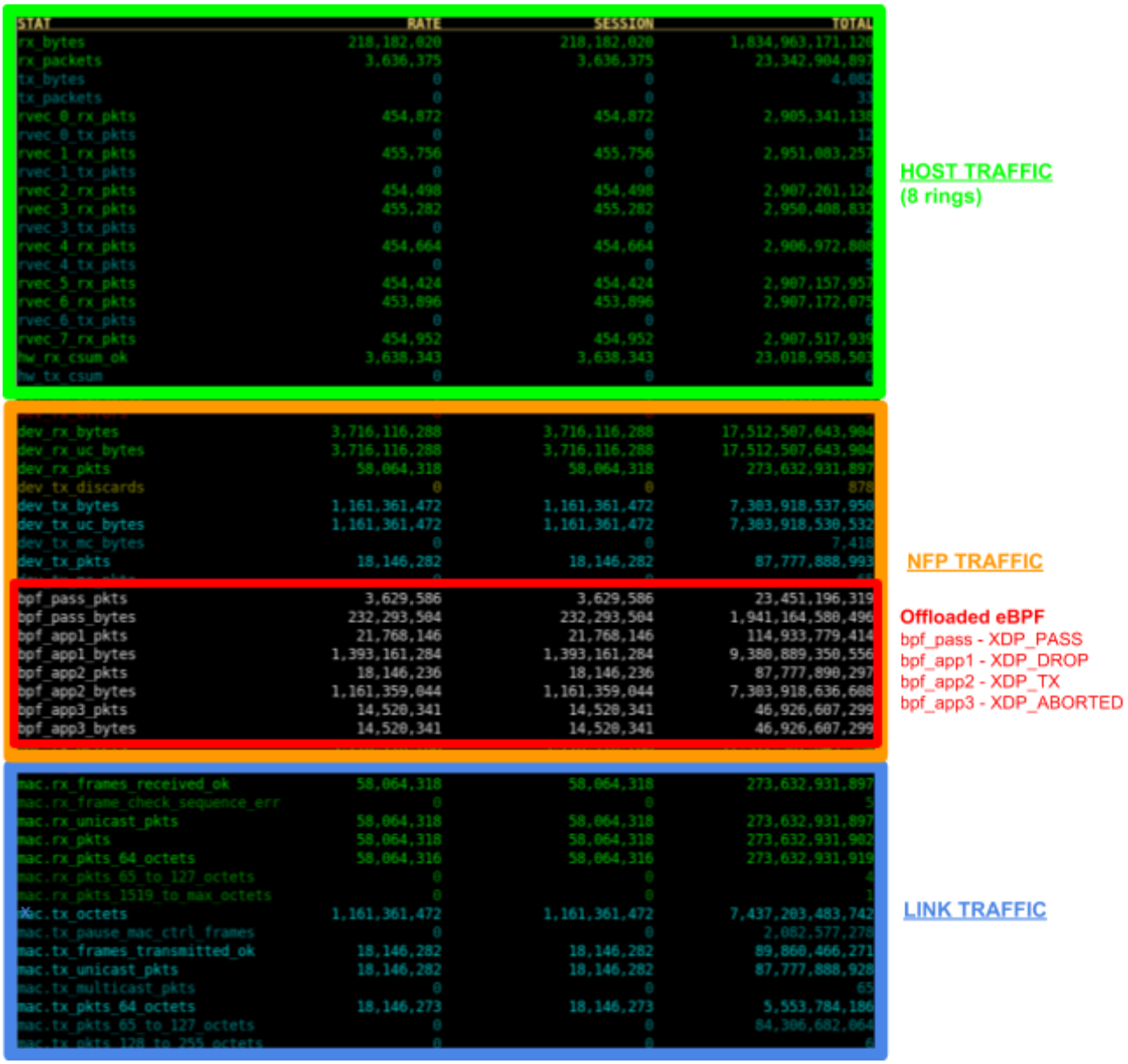
```
$ clang --version
   clang version 6.0.0
```

Please consult the relevant instructions available at https://apt.llvm.org if you need to update to `clang-6.0` or higher on a different distribution. Further instructions are also available in the Appendix.

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, in an easy-to-read fashion. It can be used as follows.

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



Offloading a basic eBPF program

If you successfully validated the steps from the previous section, your environment should be ready for performing eBPF offload. This section provides the steps for offloading a basic example program to the Agilio CX SmartNIC.

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 name).

```
# ip 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 coming to the chosen interface 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” code names are related to those used in cls_bpf for historical reasons.

Advanced programming

Maps

The NFP hardware has full ownership of offloaded maps. The host can query the map using the inbuilt kernel map lookup calls which are subsequently relayed to the NFP hardware.

Map types such as the PER_CPU variations are impractical on the NFP due to the large number of cores present therefore they are not supported. A list of supported map types can be seen in the Kernel version support section. The NFP currently has a maximum limit of 64 bytes per record (key bytes + value bytes).

Atomic writes

Since Kernel 4.17, map updates are supported by our driver. As of this writing, our public firmware does not contain map update support from the datapath, but this is available on request. Map updates can still take place from user space, for example with bpftool, see related section. Our public firmware currently supports 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_count = {
        .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_count, &key);
        if (!count)
                return XDP_DROP;
        __sync_fetch_and_add(count, 1);
        return XDP_DROP;
}
```

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.

int bpf_perf_event_output(struct pt_reg *ctx, struct bpf_map *map, u64 flags,
                          void *data, u64 size)
      Write raw data blob into a special BPF perf event held by map of type
      BPF_MAP_TYPE_PERF_EVENT_ARRAY. This perf event must have the following
      attributes: PERF_SAMPLE_RAW as sample_type, PERF_TYPE_SOFTWARE as type, and
      PERF_COUNT_SW_BPF_OUTPUT as config.
      The value to write of size, is passed to eBPF stack and pointed by data.
      For eBPF hardware offload, flags encompass two things:
            The 32 higher bits are used to indicate the number of bytes from
            context (i.e. from the packet) that will be dumped.
            The 32 lower bits must be set to BPF_F_CURRENT_CPU.

      For example, if flags are set to (0x10 << 32 | BPF_F_CURRENT_CPU), then in
      addition to size bytes of data, the first 16 (0x10) bytes of the packet
      will be dumped.
```

```
	The context of the program ctx needs also be passed to the helper.
	On user space, a program willing to read the values needs to call
	perf_event_open() on the perf event and to store the file descriptor into
	the map. This must be done before the eBPF program can send data into it.
	An example is available in file samples/bpf/trace_output_user.c in the
	Linux kernel source tree (the eBPF program counterpart is in
	samples/bpf/trace_output_kern.c).
	Data can be: only custom structs, only the packet payload, or a combination
	of both.
	Return: 0 on success, or a negative error in case of failure.
```

RX RSS Queue

The NFP allows for the offloaded eBPF program to choose the RSS queue for transferring the packets up to the host. For example, in the program below, all receiving packets will be placed onto queue 1. This can obviously be extended using hashing algorithms to provide optimized queue distributions for incoming network traffic.

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

int xdp_prog1(struct xdp_md *xdp) {
	xdp->rx_queue_index = 1;

	return XDP_PASS;
}
```

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).

Libbpf

In kernel 4.18 and newer, libbpf will offload the file if an `ifindex` is passed to `bpf_prog_load_xattr()` and if the hardware flag is set. See the Appendix for an example.

bpftool

bpftool is a user space utility used for introspection and management of eBPF objects (maps and programs).

Fedora 28 Installation

```
# yum install bpftool
```

Ubuntu 18.04 Installation

A compiled binary for Ubuntu has been made available as a Debian (.deb) package in the "*Agilio eBPF Software*" knowledge base section of our SmartNICs support website (https://help.netronome.com) which contains NFP binutils support.

Other Distributions Installation

Follow the steps in the Appendix if a binary is not available for your distribution.

Using bpftool

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
27: xdp  tag b722a8b5b9e9be25 dev ens4np0
      loaded_at Jun 12/13:20  uid 0
      xlated 112B  jited 392B  memlock 4096B  map_ids 31
```

And you could dump the instructions for this program:

```
# bpftool prog dump xlated id 27
   0: (b7) r1 = 0
   1: (63) *(u32 *)(r10 -4) = r1
   2: (bf) r2 = r10
   3: (07) r2 += -4
   4: (18) r1 = map[id:31]
   6: (85) call 0x0#1725914768
   7: (b7) r1 = 1
   8: (15) if r0 == 0x0 goto pc+3
   9: (b7) r1 = 1
  10: (c3) lock *(u32 *)(r0 +0) += r1
  11: (b7) r1 = 2
  12: (bf) r0 = r1
  13: (95) exit
```

The JIT NFP code can be dumped when bpftool is built against the latest version of binutils-dev (v2.31). The Debian (.deb) package we provide on our website does have support for dumping these JIT-ed NFP instructions.

```
# bpftool prog dump jited id 27
   0:  .0  immed[gprB_6, 0x3fff]
   8:  .1  alu[gprB_6, gprB_6, AND, *l$index1]
  10:  .2  immed[gprA_2, 0x0], gpr_wrboth
  18:  .3  immed[gprA_3, 0x0], gpr_wrboth
```

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
```

The output for perf event maps can also be displayed using bpftool:

```
# bpftool map event_pipe id 29

== @26945.050728686 CPU: 10 index: 10 =====
00 15 4d 12 1d 79 02 00  00 00 00 00 08 00 45 00
00 2e 00 00 00 00 40 06  66 c4 0a 00 00 03 0a 00
00 00 00 00
```

Debugging eBPF

This section is not strictly about eBPF offload, but provides some hints about how to debug eBPF programs, or for troubleshooting when the program is being 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. Here comes a brief descriptions of additional tools that can turn useful as well.

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 hexadecimal 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));
```

Troubleshooting

Console error (on offload attempt)	Description
`Note: 8 bytes struct bpf_elf_map fixup performed due to size mismatch!`	This is just a notification generated by iproute2 for all eBPF programs. It can be ignored.
`Map object ‘name' rejected: Operation not supported (95)!`	Check that a eBPF offload compatible driver and firmware have been installed. (see section Firmware and Driver)
`Map object ‘name’ rejected: Invalid argument (22)!`	The kernel, iproute or firmware installed does not support the map type. (see section Kernel version support)
`Offload device mismatch between prog and map`	Check that iproute2 version is newer than v4.16. (see section iproute2 utilities)
`RTNETLINK answers: Device or resource busy`	There may already be a XDP program loaded on that particular mode. Unload the existing program, or if using `ip link`, use `-force` option to forcefully load the new program.

`processed 3032 insns (limit 131072), stack depth 0`
 	
`Error fetching program/map!`
	Check `dmesg` command for further information.
	
	Program may be too large for NFP.

`Map object 'arr4' rejected: Cannot allocate memory (12)!`
`- Type: 2`
`- Max elems: 4194305`
`- Flags: 0`
	Check `dmesg` command for further information.
	
	The NFP does not have enough memory for the eBPF map, there may be too many elements within this map or an existing map may have already consumed the available memory.
	
	Note: When a eBPF program is removed, the Linux kernel does not immediately remove the map, it is instead removed several seconds later during garbage collection. A brief wait may be required between replacing programs with larger maps.

`[nfp] unsupported function id: X`	NFP does not support the eBPF helper function. Check Kernel version support to ensure your kernel can support the helper. Also check our support website for the latest firmware.

`Error: nfp: Insufficient number of TX rings w/ XDP enabled.`
 	
`(Driver mode only)`
	There are no enough available queues for XDP. Queues may be freed by reducing the number pre-allocated to the netdev using ethtool -L.
	
	(see section Setting up rings and affinities)

Appendix

Kernel Installation from source

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 -r
```

bpftool installation from kernel sources

Follow the steps below to install bpftool 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 (manual pages).

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

2. Download the kernel sources and compile the program and the documentation.

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

3. Install them on the system.

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

Clang Installation on Ubuntu 16.04

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
```

Offloading a XDP program using libbpf calls

This example shows how a eBPF program can be offloaded to the NFP using userspace libbpf calls (introduced in kernel 4.18). For driver mode, ifindex should be set to 0, for offload it should be set to the NFP interface index.

```
#include <linux/bpf.h>
#include <linux/if_link.h>
#include "bpf/libbpf.h"

int main(void)
{
	struct bpf_prog_load_attr prog_load_attr = {
		.prog_type	= BPF_PROG_TYPE_XDP,
	};
	int prog_fd;
	static int ifindex;
	static __u32 xdp_flags;
	struct bpf_object *obj;

	ifindex = 3;
	prog_load_attr.file = "file.o";
	prog_load_attr.ifindex = ifindex; /* set offload dev ifindex */
	xdp_flags |= XDP_FLAGS_HW_MODE;	  /* set HW offload flag */

	if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
		return 1;

	if (!prog_fd) {
		printf("error loading file\n");
		return 1;
	}

	if (bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
		printf("link set xdp fd failed\n");
		return 1;
	}
	return 0;
}
```

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 Sample Apps

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

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 | help.netronome.com