

Most of our customers use our service for their real time applications with very stringent performance SLAs, and we continuously invest in performance enhancements to make our service faster. One of the technologies we have invested in is eBPF for accelerating communication between various microservices in our backend. Since there are very few articles that comprehensively show how to use eBPF, we decided to share what we learned with the broader community. In this two part blog series we share in detail how we leverage eBPF for network acceleration and some interesting observations we made along the way.

This blog post specifically focuses on enabling applications to transparently bypass the TCP/IP stack using eBPF when those applications are on the same host. This is especially valuable for cloud native applications that are built using the microservices which spend a significant time processing network requests over RPCs (e.g. [gRPC](#) from the Cloud Native Computing Foundation) and/or other REST APIs. In a widely acclaimed [published work](#) researchers at Cornell have shown that microservices communication patterns make network acceleration an important aspect to consider in cloud environments. In distributed environments with latency-sensitive services where even a small improvement in tail latency is significant, network acceleration can provide a big boost in performance.

Overview of BPF and eBPF

We all have likely used the eponymous tcpdump tool to troubleshoot network or protocol development problems. The tool was [written](#) by Lawrence Berkeley National Laboratory as a means to troubleshoot the congestion collapse in the ARPAnet and they required a fast and efficient method for packet filtering for [observability](#). The requirement for a flexible and fast filtering mechanism led to the development of the Berkeley Packet Filter (BPF). BPF allowed user defined filters to be translated into instructions that ran inside a simple VM with a small register set, within the kernel, and specified which subset of network packets are to be rejected or accepted. Safety features were built in the instruction set (e.g. no unbounded loops to guarantee bounded completion, etc.) so as not to crash the kernel. One can actually use tcpdump to see the BPF instruction set in action – just use tcpdump with the -d option on an interface to get the BPF instructions for a filter (here we see the instructions for capturing UDP packets destined to DNS port 53):

```
tcpdump -i enp0s3 udp dst port 53 -d (000) ldh [12]  
(001) jeq #0x86dd jt 2 jf 6
```

:
(013) jeq #0x35 jt 14 jf 15

The above BPF, subsequently termed as the “classic” BPF, was extended to have an enhanced instruction set, new features including support for hooking at multiple events in the kernel, actions other than just packet filtering, a just-in-time assembler to increase performance, and a bytecode optimizer and verifier for the code to be injected in the kernel (see details [here](#)). The result is a general packet filter framework that can be used to inject BPF programs in the Linux kernel to extend its functionality during runtime. This enhanced form is known as extended BPF, or eBPF.

eBPF usage has taken off dramatically in recent years. eBPF is being used extensively for [observability](#) using kernel tracing (kprobes/tracing) since code in the kernel runs blazingly fast (no context switch involved) and is more accurate since it is event based. Additionally, eBPF is being used in several environments where traditional security monitoring and access control based on IP address is not sufficient (e.g. in container based environments such as Kubernetes). In Fig. 1, one can see the various hooks in the Linux kernel where an eBPF program can be hooked for execution.

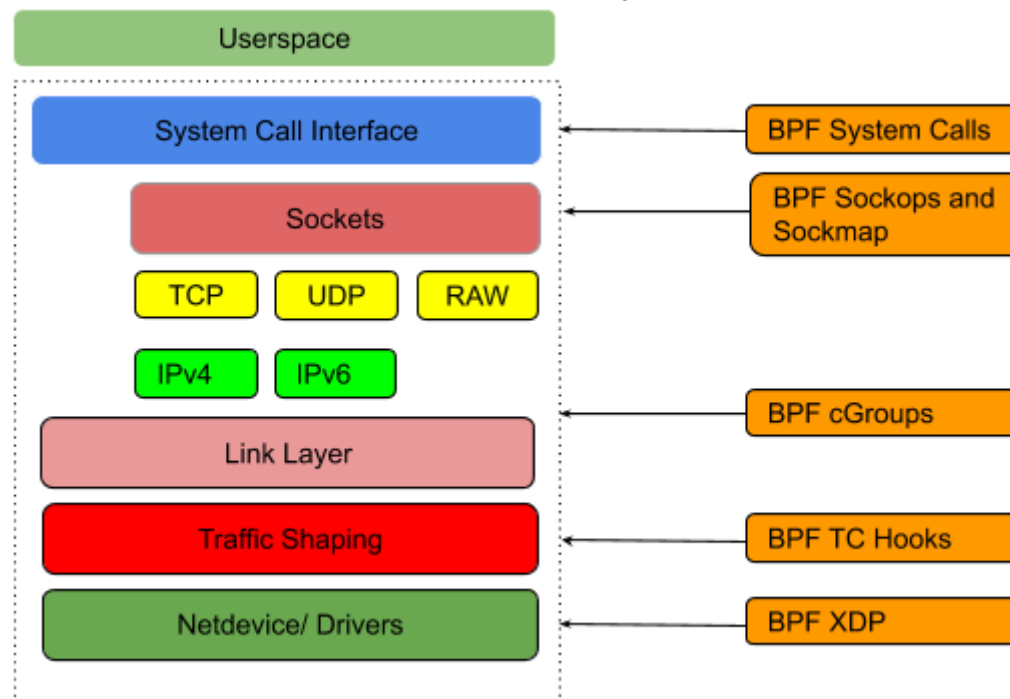


Fig.1 BPF Kernel Hooks

Using eBPF for Network Acceleration

We will now do a deep dive into the mechanics of socket data redirection using the various features available in eBPF. The full code is available [here](https://cyral.com/blog/how-to-ebpf-accelerating-cloud-native/), feel free to

download and play with it. We provide detailed instructions on how to set up your Linux environment for eBPF development.

Typically, eBPF programs have two parts to them:

- kernel space component, where decision making or data collection needs to happen based on some kernel events, such as packet rx on the nic, a system call spawning a shell, etc.
- a user space component, where one can access data written by the kernel code in some shared data structure (maps, etc.).

The code we focus on explaining is the kernel space component, and we are using the [bpftool](#) command line tool to load the code into the kernel and later unload it.

The Linux kernel supports different types of eBPF programs that can each be attached to specific hooks (see Fig. 1) available in the kernel. These programs execute when events associated with those hooks get triggered e.g. a system call

such as `setsockopt()` is made, network driver hook [XDP](#) just after DMA of the packet buffer descriptor, etc.

All the types are enumerated in the UAPI [bpf.h](#) header file with the user facing definitions required for an eBPF program. In this blog post we are interested in the eBPF programs of the type `BPF_PROG_TYPE_SOCKET_OPS` and `BPF_PROG_TYPE_SK_MSG` which allow us to hook up our BPF program to socket operations e.g. when a TCP connect event takes place, upon `sendmsg` call of a socket, etc. The `SK_MSG` is the one which performs the socket data redirect.

For this blog post, we will write the eBPF code in C and compile it with the LLVM Clang frontend to generate the ELF bytecode, to inject in the kernel, for the bpf architecture. We will take a bottoms up approach, diving into the code which actually performs the socket data redirect, and moving upward to walk through all the code operations.

Performing the socket data redirect

The SK_MSG program executes upon a sendmsg call on a socket and must be attached to a socket map, specifically BPF_MAP_TYPE_SOCKMAP or BPF_MAP_TYPE_SOCKHASH. These maps are key value stores where the value can only be a socket. Once the map has the SK_MSG program attached, all the sockets in the map inherit the SK_MSG program which gets executed upon any writes on the sockets.

```
__section("sk_msg")
int bpf_tcpip_bypass(struct sk_msg_md *msg)
{
    struct sock_key key = {};
    sk_msg_extract4_key(msg, &key);
    msg_redirect_hash(msg, &sock_ops_map, &key, BPF_F_INGRESS);
    return SK_PASS;
}
char ____license[] __section("license") = "GPL";
```

The code above is placed in the sk_msg section of the ELF object code using the compiler section attribute. This section is what tells the loader about the BPF

program type which determines where the program can be attached in the kernel and what in-kernel helper functions it can access.

`msg_redirect_hash()` is a wrapper around the BPF helper function `bpf_msg_redirect_hash()`. The helper functions cannot be accessed directly and must be accessed through predefined helpers of the form `BPF_FUNC_msg_redirect_hash` since the kernel verifier for BPF programs only allows calls to these predefined helpers from UAPI [linux/bpf.h](#) defined in *[‘enum bpf_func_id’](#)* (see the [code](#) for the macro definition). This indirection allows the BPF backend to emit error when it sees a call to a global function or to an external symbol.

The `msg_redirect_hash()` takes pointer to `sk_msg_md`, which is the user accessible metadata for the `SK_MSG` packet hook, as the first argument, the sockhash map to which this program is attached, a key which will be used to index into the map, and the flag which tells the api where to redirect the data whether to the rx queue or the egress tx queue of the socket retrieved from the sockhash map. Thus, when `sendmsg` is performed on a socket that has this program attached to it, the kernel executes this program. Then the `bpf_socket_redirect_hash()` uses the key, which identifies the destination socket

and is derived from the metadata of the message, to get access to the socket from the sockhash map and redirects the packet data to the appropriate queue of the socket depending on the flag (only defined `BPF_F_INGRESS` for rx queue or 0 is meant to be `EGRESS` for the tx queue).

Populating the sockhash map

Now let's turn our attention as to how we can populate the sockhash map used by the `SK_MSG` program. We want this sock map to be populated when a TCP connection event occurs (but before the `SK_MSG` program runs) so that data can be redirected immediately after the connection is established. Now it is the second eBPF program type `SOCK_OPS` which gets invoked upon TCP events such as connection establishment, tcp retransmit, etc. This helps capture the socket details when the TCP connection is established.

In the code snippet below, we create this program which gets triggered on socket operations, and handles the event for active (source socket sending SYN) and passive (destination socket responding with ACK for the SYN) TCP connections.

```
__section("sockops")
int bpf_sockops_v4(struct bpf_sock_ops *skops)
{
    uint32_t family, op;
    family = skops->family;
    op = skops->op;
    switch (op) {
    case BPF SOCK OPS PASSIVE_ESTABLISHED_CB:
    case BPF SOCK OPS ACTIVE_ESTABLISHED_CB:
        if (family == 2) { //AF_INET
            bpf_sock_ops_ipv4(skops);
        }
        break;
    default:
        break;
    }
    return 0;
}
char ____license[] __section("license") = "GPL";
int _version __section("version") = 1;
```

eBPF sockops programs reside in the ELF section sockops, hence the code is placed in the sockops section using the compiler section attribute. The socket operations enum value from the bpf_sock_ops struct is used to handle the TCP event along with identifying the source of the event. When the TCP connection event occurs, we call the following method to program the BPF sockhash map data structure which will be accessed from the SK_MSG program:

```
static inline
void bpf_sock_ops_ipv4(struct bpf_sock_ops *skops)
{
    struct sock_key key = {};
    sk_extractv4_key(skops, &key);
    int ret = sock_hash_update(skops, &sock_ops_map, &key, BPF_NOEXIST);
    printk("<<< ipv4 op = %d, port %d --> %d ",
        skops->op, skops->local_port, bpf_ntohl(skops->remote_port));
    if (ret != 0) {
        printk("FAILED: sock_hash_update ret: %d ", ret);
    }
}
```


In the above code, we are using the `sock_hash_update()` wrapper for accessing the BPF helper function `bpf_sock_hash_update()` to store the reference to the socket, which creates the TCP event, in the sockhash map data structure `sock_ops_map`. Thus, for a fully established TCP connection between two apps on the same host we store the reference to the source socket (app initiating the active TCP connection) as well as to the destination socket (app making the passive TCP connection by responding to the active TCP connection). Now the `SK_MSG` program, described above, can reference these values to specify which socket to redirect to via the `bpf_sk_redirect_hash()` calls.

Injecting eBPF in the Kernel

In this blog, we will use the (very useful!) `bpftool` command line tool to inject the BPF bytecodes into the kernel. It is a user-space debug utility that can also load eBPF programs, create and manipulate maps, and collect information about eBPF programs and maps. It is part of the Linux kernel tree and is available at [tools/bpf/bpftool](https://tools.bpf.dev/bpftool).

To inject the eBPF code in the kernel, we need to generate the byte code for the eBPF VM in the kernel. We will use the LLVM Clang frontend to compile the SOCK_OPS and SK_MSG code to get the object code:

```
clang -O2 -g -target bpf -I/usr/include/linux/ -I/usr/src/linux-headers-5.0.0-23/i  
-c bpf_sockops_v4.c -o bpf_sockops_v4.o  
clang -O2 -g -target bpf -I/usr/include/linux/ -I/usr/src/linux-headers-5.0.0-23/i  
-c bpf_tcpip_bypass.c -o bpf_tcpip_bypass.o
```



Let's load and attach the BPF object code for the SOCK_OPS program.

```
sudo bpftool prog load bpf_sockops_v4.o "/sys/fs/bpf/bpf_sockops" type sockops
```

- This loads the object code in the kernel (not yet attached to a hook though)

- The loaded code is pinned to a [BPF virtual filesystem](#) for persistence so that we get a handle to the program to be used later to access the loaded program. (Mount the BPF virtual filesystem using: `mount -t bpf bpf /sys/fs/bpf/`)
- By default, bpftool will create new maps as [declared](#) in the ELF objects being loaded (in this case the `sock_ops_map`)

```
sudo bpftool cgroup attach "/sys/fs/cgroup/unified/" sock_ops pinned "/sys/fs/bpf/
```



- This attaches the loaded SOCK_OPS program to the cgroup
- This is attached to cgroup so that the program applies to all sockets of all tasks placed in the cgroup (when using [cgroups v2](#), systemd automatically creates a mount point at `/sys/fs/cgroup/unified`)

Now that the object code is loaded and attached to the the SOCK_OPS hook, we need to find the id of the map used by the SOCK_OPS program which can be used to attach the SK_MSG program:

```
MAP_ID=$(sudo bpftool prog show pinned "/sys/fs/bpf/bpf_sockops" | grep -o -E 'map  
sudo bpftool map pin id $MAP_ID "/sys/fs/bpf/sock_ops_map"
```

Moving on, let's load and attach the BPF object code for the SK_MSG program.

```
sudo bpftool prog load bpf_tcpip_bypass.o "/sys/fs/bpf/bpf_tcpip_bypass" map name
```

This loads the SK_MSG object code in the kernel, pins it to the virtual filesystem at `/sys/fs/bpf/bpf_tcpip_bypass` and reuses the existing map `sock_ops_map` pinned at `/sys/fs/bpf/sock_ops_map`.

```
sudo bpftool prog attach pinned "/sys/fs/bpf/bpf_tcpip_bypass" msg_verdict pinned
```


- This attaches the loaded SK_MSG program to the sockhash map sock_ops_map pinned in the virtual filesystem, so now any socket entry in the map will have the SK_MSG program executed

Once the programs are loaded and attached to their respective hooks, we can use the bpftool to list the loaded programs:

```
sudo bpftool prog show
```

Output:

```
15: sock_ops name bpf_sockops_v4 tag 73add043b61539db gpl  
loaded_at 2020-04-06T11:31:08-0700 uid 0
```

```
19: sk_msg name bpf_tcpip_bypass tag 550f6d3cfcae2157 gpl  
loaded_at 2020-04-06T11:31:08-0700 uid 0
```

We can also get the details of the sockhash map and its contents (-p is for prettyprint):

```
sudo bpftool -p map show id 13
```

Output:

```
{  
  "id": 13,  
  "type": "sockhash",  
  "name": "sock_ops_map",  
  "flags": 0,  
  "bytes_key": 24,  
  "bytes_value": 4,  
  "max_entries": 65535,  
  "bytes_memlock": 0,  
  "frozen": 0  
}
```

```
sudo bpftool -p map dump id 13
```

Output:

```
[{  
  "key":  
  ["0x7f", "0x00", "0x00", "0x01", "0x7f", "0x00", "0x00", "0x01", "0x01", "0x00", "  
  ],  
  "value": {  
    "error": "Operation not supported"  
  }  
}]
```

```
    }, {  
      "key":  
      ["0x7f", "0x00", "0x00", "0x01", "0x7f", "0x00", "0x00", "0x01", "0x01", "0x00", "  
    ],  
      "value": {  
        "error": "Operation not supported"  
      }  
    }  
  }  
]
```

The error message is shown because the sockhash map does not support looking up its values from user space.

For a quick datapath verification of the sockhash map getting populated and being looked up by the SK_MSG program, we can use SOCAT to spin up a TCP listener and use netcat to send a TCP connection request. The SOCK_OPS program will print logs, when it gets a TCP connection event, which can be looked up from kernel's tracing file trace_pipe:

```
# start a TCP listener at port 1000, and echo back the received data
```

```
sudo socat TCP4-LISTEN:1000,fork exec:cat
```

```
# connect to the local TCP listener at port 1000
```

```
nc localhost 1000
```

```
sudo cat /sys/kernel/debug/tracing/trace_pipe
```

Output:

```
nc-1840 [000] .... 806.396867: 0: <<< ipv4 op = 4, port 41350 --> 1000
```

```
nc-1840 [000] ..s1 806.396886: 0: <<< ipv4 op = 5, port 1000 --> 41350
```

Unloading the eBPF programs

```
sudo bpftool prog detach pinned "/sys/fs/bpf/bpf_tcpip_bypass" msg_verdict pinned
```

This detaches the SK_MSG program from the sockhash map and unpins it from the virtual filesystem causing it to be cleaned up due to [reference counting](#).

```
sudo bpftool cgroup detach "/sys/fs/cgroup/unified/" sock_ops pinned "/sys/fs/bpf/  
sudo rm "/sys/fs/bpf/bpf_sockops"
```



- This detaches the SOCK_OPS program from the cgroup and unpins it from the virtual filesystem causing it to be cleaned up due to reference down counting.

```
sudo rm "/sys/fs/bpf/sock_ops_map"
```

This deletes the sock_ops_map

Conclusion

We ain't done yet! Ultimately this is all means to an end, and in the next blog post we will share the actual performance results and some very interesting lessons we learned about how it compares with TCP benefits from using Nagle's algorithm. Ultimately, we hope this blog post is useful for those trying to figure out how to use eBPF for network acceleration. Feel free to contact us at product@cyral.com if you have any questions. Also, if you like working on these technologies, come join us!

Appendix

To prepare a Linux development environment for eBPF development, various packages and kernel headers need to be installed. You can build on any Linux kernel with eBPF support. We have used Ubuntu Linux 18.04 with kernel 5.3.0-40-generic.

Ubuntu Linux

Follow the following steps to prepare your development environment:

1. Install Ubuntu 18.04

```
sudo apt-get install -y make gcc libssl-dev bc libelf-dev libcap-dev clang gcc-mul
```



```
sudo apt-get install iproute2
```

2. Download the Linux kernel source

1. You will need to update source URIs in `/etc/apt/source.list`
2. Perform the following:


```
sudo apt-get update sudo apt-get source linux-image-$(uname -r)
```

If it fails to download the source, try:

```
sudo apt-get source linux-image-unsigned-$(uname -r)
```

More information on [Ubuntu wiki](https://wiki.ubuntu.com/Kernel/KernelSources).

3. We will use the UAPI `$kernel_src_dir/include/uapi/linux/bpf.h` in the eBPF code
4. Compile and install bpftool from source. It is not yet packaged as part of the standard distributions of Ubuntu.
 1. `cd $kernel_src_dir/tools/bpf/bpftools`
 2. `make`

3. make install

5. You might also need to install libbpf-dev