eBPF Bumblebee Tutorial

Objective : Using BumbleBee eBPF to collect low-level data in kernel level. eBPF can run as sandboxed programs in OS kernel safely and extending capabilities of the kernel without changing the kernel source code or load kernel modules. eBPF program can attach to different events such as Kprobes, Uprobes, Tracepoints, Network Packets, Perf Events and etc..

1. Spin up 1 Linux server of your choice & Upgrade/Downgrade Kernel to 5.4.0 and above
2. Install Docker Engine
3. Install & Configure BumbleBee

1) Steps to downgrade or upgrade Kernel in Ubuntu

a) After your Linux(Ubuntu) server is up and running, please check your kernel version via "uname -r"

```
root@iZt4n2lgtoxiw4wcmguisoZ:~# uname -r
5.8.0-050800-generic
```

b) Let's download the ubuntu kernel.sh

- wget https://raw.githubusercontent.com/pimlie/ubuntu-mainline-kernel.sh/master/ubuntu-mainline-kernel.sh

c) Install ubuntu mainline kernel

- sudo install ubuntu-mainline-kernel.sh /usr/local/bin/

d) Make is executable to the file

- chmod +x ubuntu-mainline-kernel.sh

e) **Search and find your desired version**

- ubuntu-mainline-kernel.sh -r | grep 5.4.0

```
root@iZt4n2lgtoxiw4wcmguisoZ:~# ubuntu-mainline-kernel.sh -r | grep 5.4.0
v5.4.0          v5.4.5          v5.4.10         v5.4.15          v5.4.20
root@iZt4n2lgtoxiw4wcmguisoZ:~# ubuntu-mainline-kernel.sh -r | grep 5.8.0
v5.7.17         v5.7.18         v5.7.19         v5.8.0           v5.8.5
```

f) Install the chosen kernel

ubuntu-mainline-kernel.sh -i v5.8.0

g) List down all the menuentry so you can input in the grub later.

- grep 'menuentry \|submenu ' /boot/grub/grub.cfg | cut -f2 -d "'"

```
root@iZt4n2lgtoxiw4wcmguisoZ:~# grep 'menuentry \|submenu ' /boot/grub/grub.cfg | cut -f2 -d "'"
Ubuntu
Advanced options for Ubuntu
Ubuntu, with Linux 5.8.0-050800-generic
Ubuntu, with Linux 5.8.0-050800-generic (recovery mode)
Ubuntu, with Linux 5.4.0-164-generic
Ubuntu, with Linux 5.4.0-164-generic (recovery mode)
Ubuntu, with Linux 5.4.0-42-generic
Ubuntu, with Linux 5.4.0-42-generic (recovery mode)
```

change the grub configuration

- vi /etc/default/grub
  from: GRUB_DEFAULT=0
  to: GRUB_DEFAULT="Advanced options for Ubuntu>Ubuntu, with Linux 5.8.0-050800-generic"

h) Proceed to update the grub and reboot the server

"update-grub" & "reboot"

2) Install Docker Engine

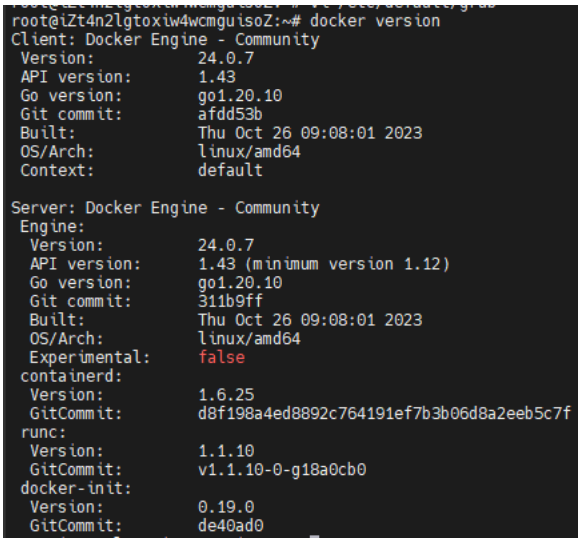
The reason why we need to install docker is because bumblebee uses the same docker-like technology to build, push and run image like how you use docker. Thus, it is required to have docker engine installed.

 # Add Docker's official GPG key:
sudo apt-get update
sudo apt-get install ca-certificates curl gnupg
sudo install -m 0755 -d /etc/apt/keyrings
curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /etc/apt/keyrings/docker.gpg
sudo chmod a+r /etc/apt/keyrings/docker.gpg

 # Add the repository to Apt sources:
```
echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
sudo apt-get update
```

Ref: https://docs.docker.com/engine/install/ubuntu/

Verification of Docker installed.



3) Install & Configure (Init, Build, Run) BumbleBee eBPF

a) Install bumblebee with your desired version (0.0.8 to 0.0.14.

curl -sL https://run.solo.io/bee/install | BUMBLEBEE_VERSION=v0.0.14 sh

b) Add the environment path

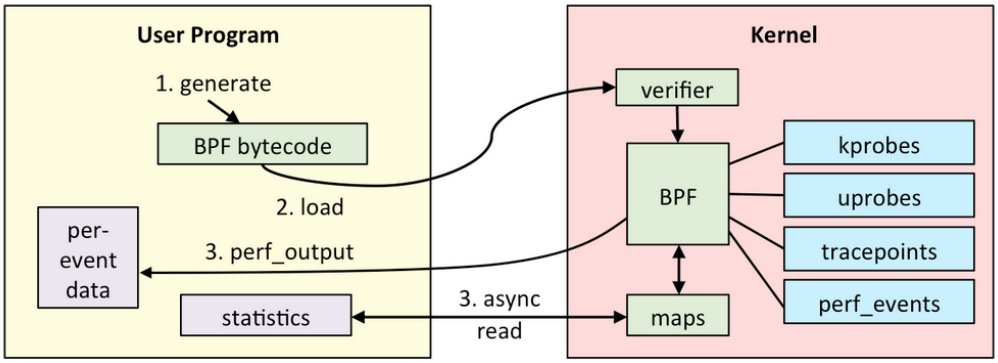
export PATH = $HOME/.bumblebee/bin:$PATH

c) Giving capabilities to eBPF

Loading eBPF programs to the kernel (`bee run` command) requires elevated privileges. You can either run `bee` as root (with sudo), or add capabilities to the binary. Adding capabilities is the preferred method, as if you run `bee run` with `sudo`, it will not find local images when you run `bee build` without sudo.

sudo setcap cap_sys_resource,cap_sys_admin+eip $(which bee)

d) As Bumblebee has make it more easier for users to implement eBPF. We only need to write everything in C in Kernel with the templates which I will show later.

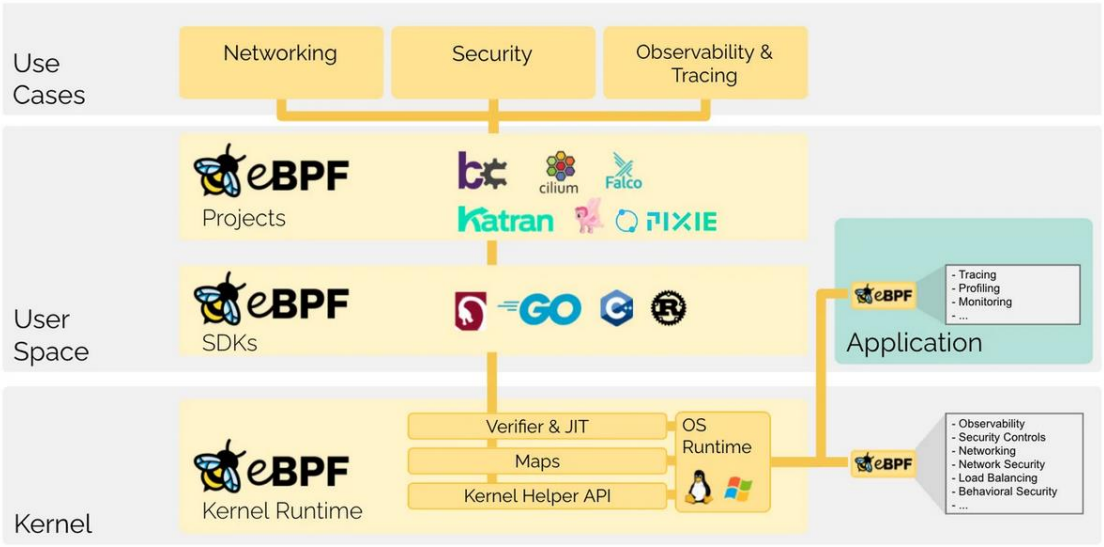


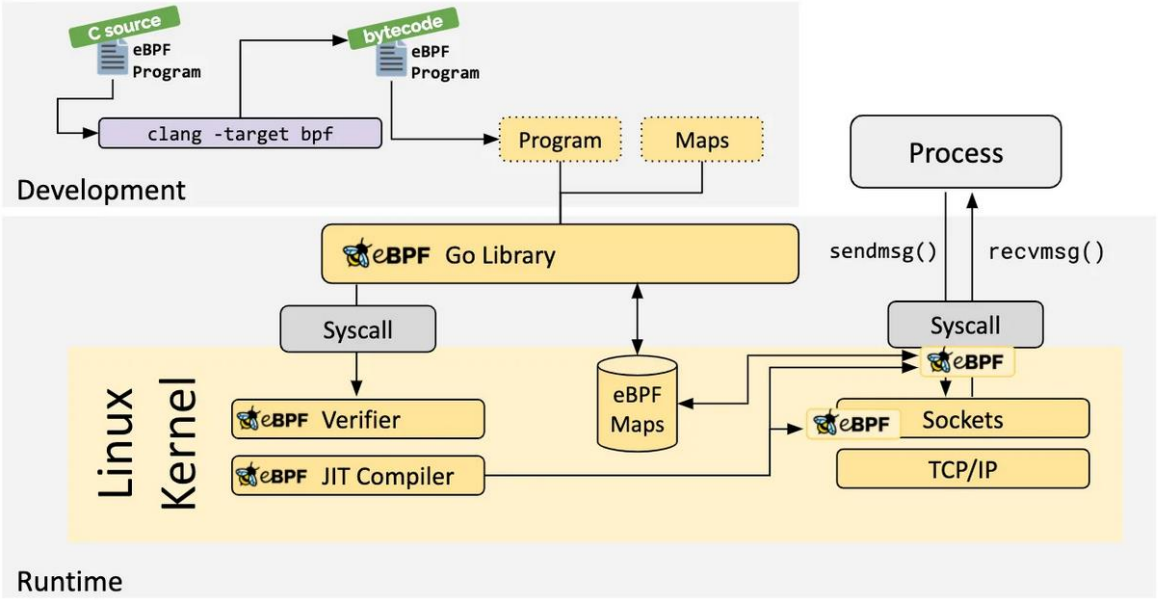
Just some background of eBPF. First generation (2020) is using BCC which requires to write code in C in kernel space and Python in userspace. BCC uses built-on Clang or LLVM compiler to compile BPF Code in runtime. This compilation can be resource intensive as CPU and RAM will utilize a lot everytime you compile.

Later Libbpf + BPF CO-RE (Compile Once – Run Everywhere) comes into picture to close the gaps of BCC especially for tracing applications and many other things BCC cant fulfil. It requires to write code in C in both Kernel and Userspace.

Finally Bumblebee came into picture to simplify everything that only requires to write C in Kernel. Covering a wide array of use cases, including next-generation networking,

observability, and security functionality. [GitHub - lizrice/ebpf-beginners: The beginner's guide to eBPF]





Why eBPF is required? Because we do not want to change kernel code. Below is the cons

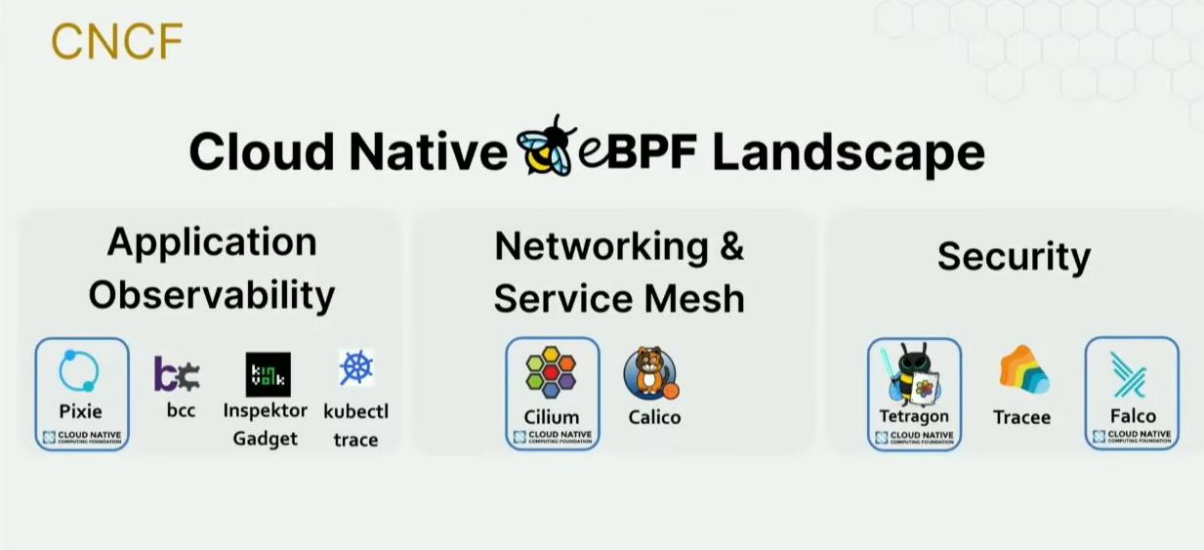
Native Support

1. Change kernel source code and convince the Linux kernel community that the change is required.
2. Wait several years for the new kernel version to become a commodity.

Kernel Module

1. Write a kernel module
2. Fix it up regularly, as every kernel release may break it
3. Risk corrupting your Linux kernel due to lack of security boundaries

Use Case of eBPF



Hyperscalers

Meta

github.com/facebookincubator/katran

Blazing fast,  **eBPF**-based L4 load-balancer used at Facebook/Meta

Smartphones

android

Android BPF loader

During Android boot, all eBPF programs located at `/system/etc/bpf/` are loaded. These programs are binary objects built by the Android build system from C programs and are accompanied by `Android.bp` files in the Android source tree. The build system stores the generated objects at `/system/etc/bpf`, and those objects become part of the system image.

Examples of eBPF in Android

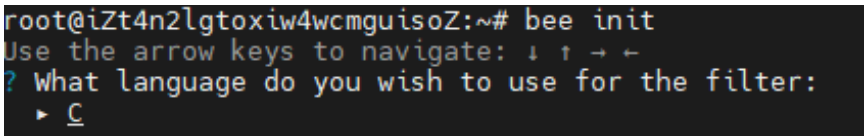
The following programs in AOSP provide additional examples of using eBPF:

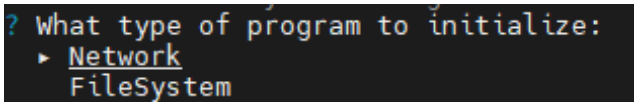
- The `netd` eBPF C program is used by the networking daemon (netd) in Android for various purposes such as socket filtering and statistics gathering. To see how this program is used, check the eBPF traffic monitor sources.
- The `time_in_state` eBPF C program calculates the amount of time an Android app spends at different CPU frequencies, which is used to calculate power.
- In Android 12, the `gpu_mem` eBPF C program tracks total GPU memory usage for each process and for the entire system. This program is used for GPU memory profiling.

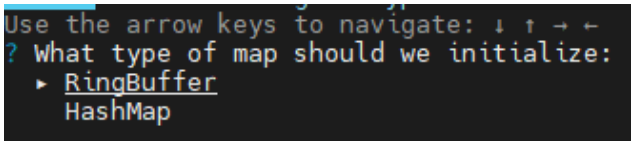
https://source.android.com/docs/core/architecture/kernel/bpf

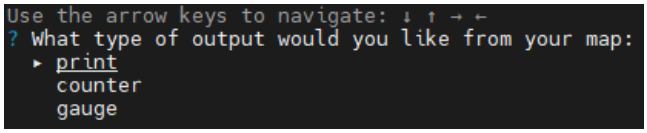
Enough said, let's do some quick hands-on on bumblebee eBPF and see what are the things we can collect.

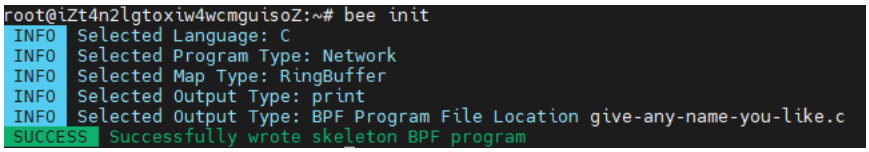
After installation of bee. You can build your own C program with "bee init" and choose the subsequent options accordingly.







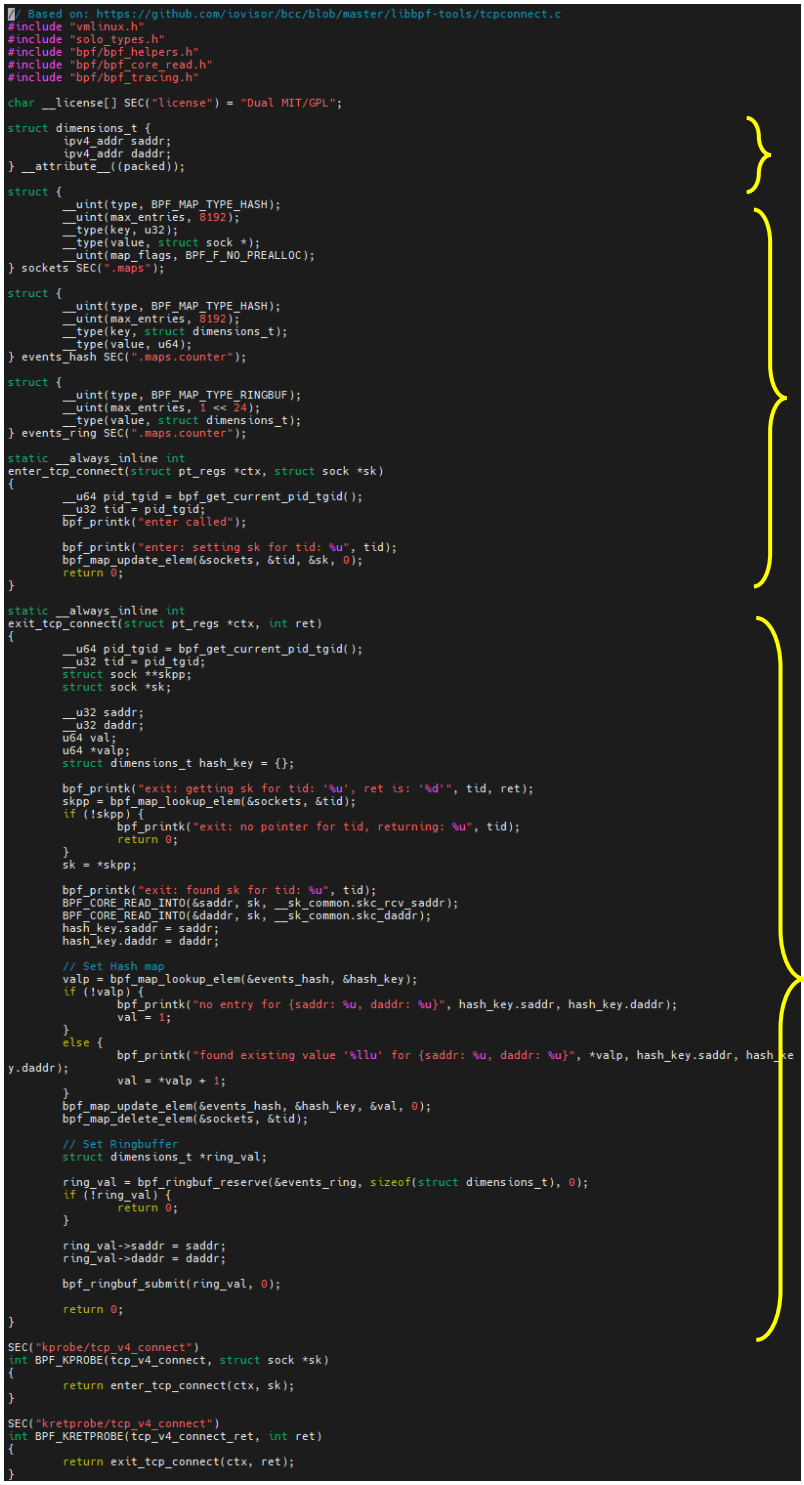




Or you could use some of the available C script in

https://github.com/solo-io/bumblebee/tree/main/examples

For instance, I will be using "tcpconnect.c" to trace the events for source address and destination address whenever any active connection is going on. Let me show you an example of the script.



```c
// Based on: https://github.com/iovisor/bcc/blob/master/libbpf-tools/tcpconnect.c
#include "vmlinux.h"
#include "solo_types.h"
#include "bpf/bpf_helpers.h"
#include "bpf/bpf_core_read.h"
#include "bpf/bpf_tracing.h"

char __license[] SEC("license") = "Dual MIT/GPL";

struct dimensions_t {
        ipv4_addr saddr;
        ipv4_addr daddr;
} __attribute__((packed));

struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __uint(max_entries, 8192);
        __type(key, u32);
        __type(value, struct sock *);
        __uint(map_flags, BPF_F_NO_PREALLOC);
} sockets SEC(".maps");

struct {
        __uint(type, BPF_MAP_TYPE_HASH);
        __uint(max_entries, 8192);
        __type(key, struct dimensions_t);
        __type(value, u64);
} events_hash SEC(".maps.counter");

struct {
        __uint(type, BPF_MAP_TYPE_RINGBUF);
        __uint(max_entries, 1 << 24);
        __type(value, struct dimensions_t);
} events_ring SEC(".maps.counter");

static __always_inline int
enter_tcp_connect(struct pt_regs *ctx, struct sock *sk)
{
        __u64 pid_tgid = bpf_get_current_pid_tgid();
        __u32 tid = pid_tgid;
        bpf_printk("enter called");

        bpf_printk("enter: setting sk for tid: %u", tid);
        bpf_map_update_elem(&sockets, &tid, &sk, 0);
        return 0;
}

static __always_inline int
exit_tcp_connect(struct pt_regs *ctx, int ret)
{
        __u64 pid_tgid = bpf_get_current_pid_tgid();
        __u32 tid = pid_tgid;
        struct sock **skpp;
        struct sock *sk;

        __u32 saddr;
        __u32 daddr;
        u64 val;
        u64 *valp;
        struct dimensions_t hash_key = {};

        bpf_printk("exit: getting sk for tid: '%u', ret is: '%d'", tid, ret);
        skpp = bpf_map_lookup_elem(&sockets, &tid);
        if (!skpp) {
                bpf_printk("exit: no pointer for tid, returning: %u", tid);
                return 0;
        }
        sk = *skpp;

        bpf_printk("exit: found sk for tid: %u", tid);
        BPF_CORE_READ_INTO(&saddr, sk, __sk_common.skc_rcv_saddr);
        BPF_CORE_READ_INTO(&daddr, sk, __sk_common.skc_daddr);
        hash_key.saddr = saddr;
        hash_key.daddr = daddr;

        // Set Hash map
        valp = bpf_map_lookup_elem(&events_hash, &hash_key);
        if (!valp) {
                bpf_printk("no entry for {saddr: %u, daddr: %u}", hash_key.saddr, hash_key.daddr);
                val = 1;
        }
        else {
                bpf_printk("found existing value '%llu' for {saddr: %u, daddr: %u}", *valp, hash_key.saddr, hash_ke
y.daddr);
                val = *valp + 1;
        }
        bpf_map_update_elem(&events_hash, &hash_key, &val, 0);
        bpf_map_delete_elem(&sockets, &tid);

        // Set Ringbuffer
        struct dimensions_t *ring_val;

        ring_val = bpf_ringbuf_reserve(&events_ring, sizeof(struct dimensions_t), 0);
        if (!ring_val) {
                return 0;
        }

        ring_val->saddr = saddr;
        ring_val->daddr = daddr;

        bpf_ringbuf_submit(ring_val, 0);

        return 0;
}

SEC("kprobe/tcp_v4_connect")
int BPF_KPROBE(tcp_v4_connect, struct sock *sk)
{
        return enter_tcp_connect(ctx, sk);
}

SEC("kretprobe/tcp_v4_connect")
int BPF_KRETPROBE(tcp_v4_connect_ret, int ret)
{
        return exit_tcp_connect(ctx, ret);
}
```

Stating events here to collect source/destination address. Sending these 2 to transport and capture

MAP – HashMap instead of RingBuffer to collect Key Value. Define this to send to userspace

Userspace – Print the outcome

Now let’s close this and run this command to execute it “bee run tcp:v1” based on your image and run 2 terminal. One will be userspace to show the result, another will be “bee run tcp:v1”

```
root@iZt4n2lgtoxiw4wcmguisoZ:~# curl www.apple.com
root@iZt4n2lgtoxiw4wcmguisoZ:~# ping www.apple.com
PING e6858.dscx.akamaiedge.net (23.75.212.211) 56(84) bytes of data.
64 bytes from a23-75-212-211.deploy.static.akamaitechnologies.com (23.75.212.211): icmp_seq=1 ttl=56
 time=3.83 ms
64 bytes from a23-75-212-211.deploy.static.akamaitechnologies.com (23.75.212.211): icmp_seq=2 ttl=56
 time=2.92 ms
^C
```