eBPF Filters | Agenda

- Tracing overview
  - Dynamic tracing
  - Static tracing
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- eBPF example
- Software stack

eBPF Filters | Agenda

- **Tracing overview**
  - **Dynamic tracing**
    - **Kernel**
    - Userspace
  - Static tracing
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- eBPF example
- Software stack

Dynamic | Kprobes

"Kprobes enables you to dynamically break into any kernel routine and collect debugging and performance information non-disruptively. You can trap at almost any kernel code address [1], specifying a handler routine to be invoked when the breakpoint is hit." - Documentation/kprobes.txt

[1] some parts of the kernel code can not be trapped, kprobes_blacklist

- Lives in kernel since version 2.6.9, 2004 (as for today, the latest release is 5.12.1)
- Enabled by default in most linux distributions (CONFIG_KPROBES=Y 👍)
- Two types - kprobe and kretprobe

Dynamic | Kprobes | kprobe

- Can be inserted on any arbitrary instruction in the kernel
- *pre_handler* is called when the probed instruction is about to execute
- *post_handler* called on successful completion, otherwise *fault_handler*
- All handlers have access to registers

```c
struct kprobe {
  const char *symbol_name;

  /* Offset into the symbol */
  unsigned int offset;

  /* Called before addr is executed */
  kprobe_pre_handler_t pre_handler;

  /* Called after addr is executed */
  kprobe_post_handler_t post_handler;

  /* Called if executing addr causes a fault */
  kprobe_fault_handler_t fault_handler;
  // ...
};
```

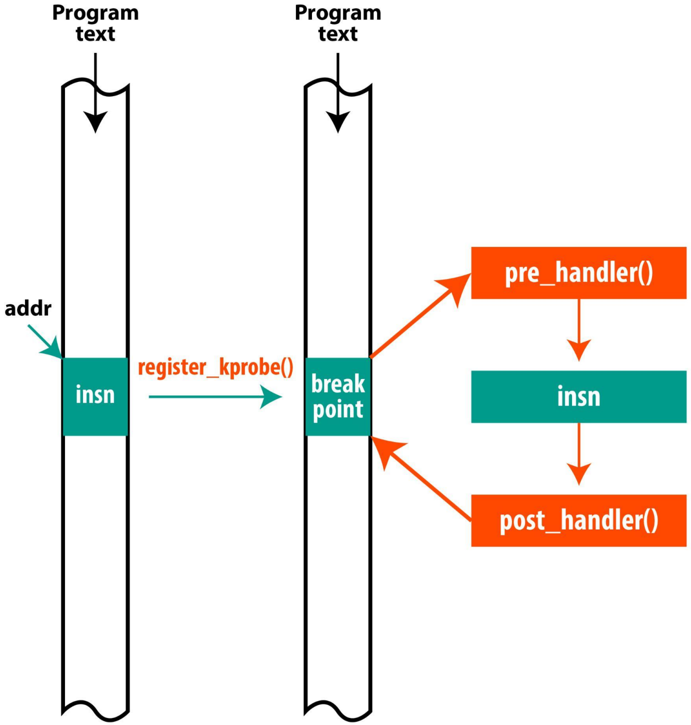
Dynamic | Kprobes | kretprobe

- Fires on function returns
- Can be also used to hook up on function entries
- Make sense if both entry and exit probes needed

```
struct kretprobe {
    // ...
    kretprobe_handler_t handler;
    kretprobe_handler_t entry_handler;
    //
    int maxactive;
    // ...
};
```

Dynamic | Kprobes | How does it work?

- register_kprobe() / unregiser_probe()
- A probed instruction is saved and replaced with a breakpoint instruction
- Instruction flow hits that breakpoint,
- The breakpoint handler check if it was installed by kprobe and executes kprobe handler
- The original instruction is then executed, wrapped with handlers



eBPF Filters | Agenda

- **Tracing overview**
  - **Dynamic tracing**
    - Kernel
    - **Userspace**
  - Static tracing
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- eBPF example
- Software stack

Dynamic | Uprobes

- In Linux since 3.5 kernel, 2012
- Similar to kprobes but for user-space processes
- Instrument user-level function entries (uprobe) and returns (uretprobe)
- File-based and system-wide
- Can be explored in action using a debugger

```
$ gdb -p 6080
(gdb) disas main
Dump of assembler code for function main:
   0x000055abb74e3db0 <+0>:  int3
   0x000055abb74e3db1 <+1>:  push   %rdi
   0x000055abb74e3db2 <+2>:  push   %r14
   0x000055abb74e3db4 <+4>:  push   %r13
```

Dynamic | Uprobes | uprobe interface

- Define a new probe by appending a line to debug file *\/sys/kernel/debug/tracing/uprobe_events*
    - p[:EVENT] SYMBOL[+offset|-offset] [FETCHARGS] format for uprobe
    - r[:EVENT] SYMBOL[+0] [FETCHARGS] format for uretprobe
    - FETCHARGS describes the data to be fetched, e.g registers, stack offsets, function arguments, etc
- The event will appear /sys/kernel/debug/tracing/events/uprobes directory
    - It's not turned on by default
    - echo 1 > events/uprobes/<probe>/enabled

Dynamic | Uprobes | Tracing Buffer

Example of tracing buffer output after planting a uprobe into bash’s main

```
# echo 'p /bin/bash:0x4245c0' > /sys/kernel/debug/tracing/uprobe_events
# echo 1 > /sys/kernel/debug/tracing/events/uprobes/enable
# cat /sys/kernel/debug/tracing/trace
#           TASK-PID   CPU#  ||||    TIMESTAMP  FUNCTION
#              | |       |   ||||       |         |
           bash-24217 [002] d... 159003.152693: p_bash_0x2fdb0: (0x5638ae66bdb0)
           bash-24227 [002] d... 159004.416583: p_bash_0x2fdb0: (0x5557446d3db0)
           bash-24237 [001] d... 159005.592744: p_bash_0x2fdb0: (0x55738a29cdb0)
           bash-24247 [003] d... 159006.888791: p_bash_0x2fdb0: (0x557add83ddb0)
      lesspipe-24335 [002] d... 159024.394264: p_bash_0x2fdb0: (0x55d909ffddb0)
```

eBPF Filters | Agenda

- **Tracing overview**
  - Dynamic tracing
  - **Static tracing**
    - **Kernel**
    - Userspace
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- eBPF example
- Software stack

Static | Tracepoints

- Used for kernel static instrumentation, originally called Kernel Markers
  - Patch merged in 2.6.32 release in 2009
- They are inserted at important predefined places in code by developers
  - Tracepoints provide a stable API, thus tools written to use tracepoints should continue working across new kernel releases
- Divided into groups, e.g *kmem, fs, sched*
  - The format is subsystem:eventname, e.g kmem:kmalloc or sched:sched_process_exec
- Disabled by default and enabled via the debugfs, same way as uprobes and kprobes
  - /sys/kernel/debug/tracing/events/kmem/kmalloc/enable

Static | Tracepoints | Arguments

Every tracepoint has its args. One way to list them is to cat the relevant format file:

```
root@ihar-UX310UQ:/# cat /sys/kernel/debug/tracing/events/sched/sched_switch/format
name: sched_switch
ID: 305
format:
        field:unsigned short common_type;       offset:0;       size:2; signed:0;
        field:unsigned char common_flags;       offset:2;       size:1; signed:0;
        field:unsigned char common_preempt_count;       offset:3;       size:1; signed:0;
        field:int common_pid;   offset:4;       size:4; signed:1;

        field:char prev_comm[16];       offset:8;       size:16;        signed:1;
        field:pid_t prev_pid;   offset:24;      size:4; signed:1;
        field:int prev_prio;    offset:28;      size:4; signed:1;
        field:long prev_state;  offset:32;      size:8; signed:1;
        field:char next_comm[16];       offset:40;      size:16;        signed:1;
        field:pid_t next_pid;   offset:56;      size:4; signed:1;
        field:int next_prio;    offset:60;      size:4; signed:1;
```

eBPF Filters | Agenda

- **Tracing overview**
  - Dynamic tracing
  - **Static tracing**
    - Kernel
    - **Userspace**
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- eBPF example
- Software stack

Static | User Statically Defined Tracepoints (USDTs)

- USDTs provide static tracepoints for programs in user-space
- Like kernel tracepoints, require developers to put instructions in code
- Work the following way:
  - At compilation a no-operation (nop) instruction is placed at the address of the USDT probe
  - This address is then dynamically changed by the kernel to a breakpoint when instrumented
- With a little bit of overhead allow instrumentation of applications in production

```c
#include <sys/sdt.h>

int main() {
  DTRACE_PROBE("hello-usdt", "probe");
}
```

```
:!readelf -n hello_usdt
```

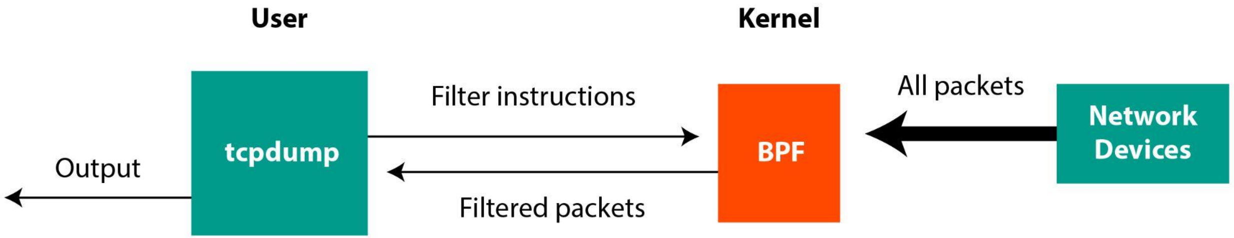
```
Displaying notes found in: .note.stapsdt
  Owner                      Data size
  stapsdt                    0x0000002e
    Provider: "hello-usdt"
    Name: "probe"
    Location: 0x00000000000005fe, Base:
    Arguments:
```

eBPF Filters | Agenda

- Tracing overview
  - Dynamic tracing
  - Static tracing
- **Berkeley Packet Filter**
  - **Classic BPF**
  - Extended BPF
- eBPF example
- Software stack

Classic BPF | Berkeley Packet Filter

- Classic BPF arrived in Linux in 1997, for the 2.1.75 kernel
- In-kernel virtual machine with a reduced set instructions run in an isolated environment
  - Two 32-bit registers: A, X and 16 x 32 bit wide misc registers: M[]
  - Instructions are capable only for filtering network packets, e.g.:
    - Load / Store from packet: **ld, st**
    - Conditional jumping targets: **jt, jf**
- Instructions are specified by end users and passed to the kernel for execution
- Performance is reached by avoiding costly copies from kernel to user-level processes



cBPF | tcpdump -d "tcp port 80"

```
(000) ldh      [12]
(001) jeq      #0x86dd          jt 2     jf 8
(002) ldb      [20]
(003) jeq      #0x6             jt 4     jf 19
(004) ldh      [54]
(005) jeq      #0x50            jt 18    jf 6
(006) ldh      [56]
(007) jeq      #0x50            jt 18    jf 19
(008) jeq      #0x800           jt 9     jf 19
(009) ldb      [23]
(010) jeq      #0x6             jt 11    jf 19
(011) ldh      [20]
(012) jset     #0x1fff          jt 19    jf 13
(013) ldxb     4*([14]&0xf)
(014) ldh      [x + 14]
(015) jeq      #0x50            jt 18    jf 16
(016) ldh      [x + 16]
(017) jeq      #0x50            jt 18    jf 19
(018) ret      #262144
(019) ret      #0
```

- **Instruction 000**
  - Loads the packet's offset 12 into accumulator, offset 12 represents an ethertype
- **Instruction 001**
  - Compares the value against **0x86dd**
  - **0x86dd** is ethertype value for **IPv6**
- **Instructions 002 and 003**
  - Checks if it's a **TCP** packet
- **Instructions 004 - 005 and 006-007**
  - Checks the source and destination ports

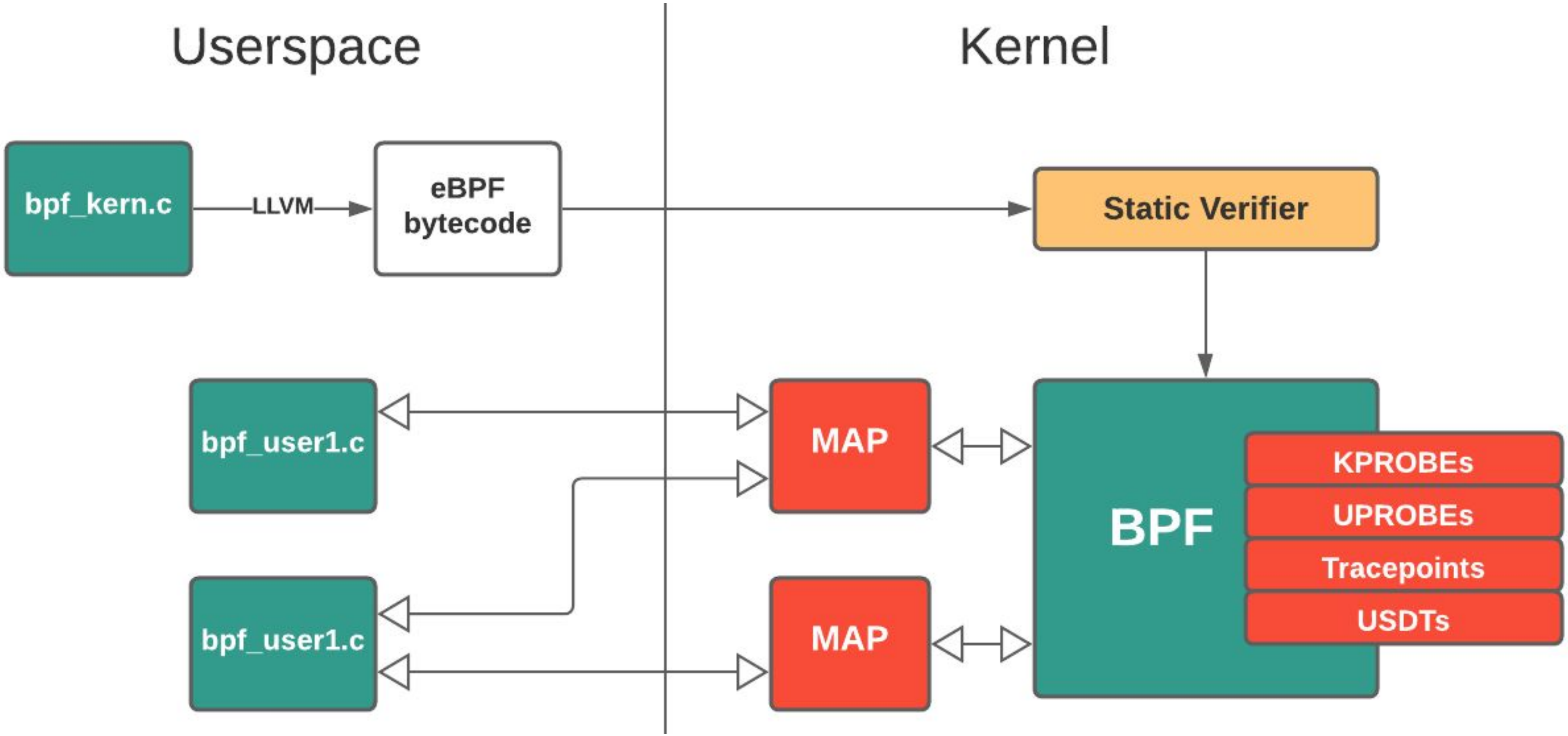
eBPF Filters | Agenda

- Tracing overview
  - Dynamic tracing
  - Static tracing
- **Berkeley Packet Filter**
  - Classic BPF
  - **Extended BPF**
- eBPF example
- Software stack

eBPF = Extended Berkeley Packet Filter | BPF

- Improved VM architecture
  - Similar to x86-64, 10 x 64-bit registers
  - JIT-compiling for programs
- Decoupled from the networking subsystem
  - **It’s possible to write and attach programs to tracepoints and kprobes**
  - Packet-filtering is implemented on top of eBPF nowadays
- Persistent maps
  - eBPF programs have ability to interact with user space via KV-storage
- Tail-calls
  - eBPF programs are limited in size, max 4096
  - “Program array” map stores references to other eBPF programs
- eBPF is **fully programmable** now

eBPF | Overview



eBPF | Program Types

- **Tracing** - KPROBE, TRACEPOINT, PERF_EVENT
- **Networking**. e.g:
  - SOCK_OPS - connection establishment, retransmit timeout
  - SOCKET_FILTER - an alternative to cBPF (tcpdump)
  - SK_SKB - access socket details (port, IP), buffer, redirect, stream parsing
- Program type determines what kernel helpers are available for calling as well as what context is provided as an argument , e.g:
  - BPF_FUNC_get_stack()
  - BPF_FUNC_override_return()
  - BPF_FUNC_ger_current_uid_pid() and many more

```
enum bpf_prog_type {
  BPF_PROG_TYPE_UNSPEC,
  BPF_PROG_TYPE_SOCKET_FILTER,
  BPF_PROG_TYPE_KPROBE,
  BPF_PROG_TYPE_SCHED_CLS,
  BPF_PROG_TYPE_SCHED_ACT,
  BPF_PROG_TYPE_TRACEPOINT,
  BPF_PROG_TYPE_XDP,
  BPF_PROG_TYPE_PERF_EVENT,
  BPF_PROG_TYPE_CGROUP_SKB,
  BPF_PROG_TYPE_CGROUP_SOCK,
  BPF_PROG_TYPE_LWT_IN,
  BPF_PROG_TYPE_LWT_OUT,
  BPF_PROG_TYPE_LWT_XMIT,
  BPF_PROG_TYPE_SOCK_OPS,
  BPF_PROG_TYPE_SK_SKB,
  BPF_PROG_TYPE_CGROUP_DEVICE,
};
```

eBPF | Maps

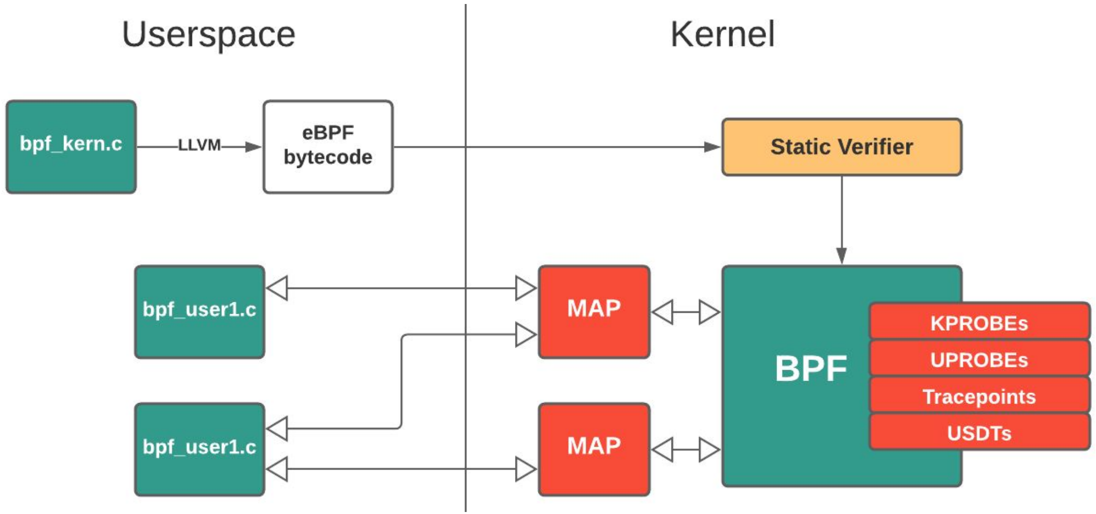
- BPF maps are key/value stores that reside in the kernel
- Can be accessed by **any** BPF program that knows about them
- Programs in user-space can access maps by using file descriptors
  - A data is copied to kernel before updating the map, this makes update operations not atomic
- Concurrent access to map elements is managed via spin locks
  - Lock access to a map’s element while you’re operating on it
- Can store different types of data
  - Hash-Table Maps (per-cpu), Array Maps (per-cpu, cgroup), Queue Maps, Stack Maps
  - Array of Maps and Hash of Maps
- Useful for keeping **state** between invocations of an BPF program

eBPF Filters | Agenda

- Tracing overview
  - Dynamic tracing
  - Static tracing
- Berkeley Packet Filter
  - Classic BPF
  - Extended BPF
- **eBPF example**
- Software stack

eBPF | Overview

- Backend
  - eBPF bytecode loaded and running in the kernel
  - Writes data to shared maps
- Loader
  - Loads backend's bytecode
  - Bytecode gets automatically unloaded when its loader terminates
- Frontend
  - User-space processes that read data from maps
- Data structures
  - Means of communication



eBPF | Socket filter | Backend

```c
struct bpf_map_def SEC("maps") map = {
    .type = BPF_MAP_TYPE_ARRAY,
    .key_size = sizeof(__u32),
    .value_size = sizeof(__u64),
    .max_entries = 32
};

SEC("socket")
int bpf_program(struct __sk_buff *skb) {
    // Only outgoing packets
    if (skb->pkt_type != PACKET_OUTGOING) return 0;
    __u32 proto, dest;
    // Check IP protocol
    proto = load_byte(skb, ETH_HLEN + IP_PROTO_OFF);
    if (proto != IPPROTO_ICMP && proto != IPPROTO_UDP) return 0;
    // Check destination address (lo)
    bpf_skb_load_bytes(skb, ETH_HLEN + IP_DEST_OFF, &dest, sizeof(dest));
    if (ntohl(dest) != 0x7f000001) return 0;
    // Update shared map value
    long *value = bpf_map_lookup_elem(&map, &proto);
    if (value) __sync_fetch_and_add(value, skb->len);
    return 0;
}
```

- Creating map requires configuration attributes for its type and size
  - **BPF_MAP_TYPE_ARRAY**
  - Elements are pre-allocated and zero initialized
  - Index size can only be 4 bytes
  - Location is in “maps” ELF-section
- **SEC** is used to place data and code specific named sections in ELF
  - ELF loader for eBPF programs looks for maps by scanning the ELF section named "map
- Program type is **BPF_PROG_TYPE_SOCKET_FILTER**
  - Base on it it has access to specific helper functions like **bpf_skb_load_bytes**
- The lookup **bpf_map_lookup_elem()** returns a ptr into the array element
  - Compiler primitive **__sync_fetch_and_add**() is used to update the value in-place to avoid data races with userspace reading the value

eBPF | Socket filter | Frontend

```c
int main(int argc, char const *argv[])
{
    struct bpf_object *obj;
    int prog_fd, map_fd, sock;
    // LOADER
    bpf_prog_load("bpf_program.o", BPF_PROG_TYPE_SOCKET_FILTER, &obj, &prog_fd);
    // Find file descriptor for shared map
    map_fd = bpf_object__find_map_fd_by_name(obj, "map");
    // Attach BPF program as a socket filter for AF_PACKET sockets
    sock = socket(AF_PACKET, SOCK_RAW, htons(ETH_P_ALL));
    setsockopt(sock, SOL_SOCKET, SO_ATTACH_BPF, &prog_fd, sizeof(prog_fd));
    // FRONTEND
    FILE *f = popen("ping -4 -c5 localhost", "r");
    for (size_t i = 0; i < 5; i++) {
        int index;
        long udp_cnt, icmp_cnt;
        index = IPPROTO_UDP;
        assert(bpf_map_lookup_elem(map_fd, &index, &udp_cnt) == 0);
        index = IPPROTO_ICMP;
        assert(bpf_map_lookup_elem(map_fd, &index, &icmp_cnt) == 0);
        printf("UDP %lld ICMP %lld bytes\n", udp_cnt, icmp_cnt);
        sleep(1);
    }
    return 0;
}
```

* When loading we specify program type as **BPF_PROG_TYPE_SOCKET_FILTER,** it will match alls **SEC**tions that start with “socket”

UDP 0 ICMP 0 bytes
UDP 46 ICMP 588 bytes
UDP 97 ICMP 980 bytes
UDP 149 ICMP 980 bytes
UDP 194 ICMP 980 bytes

eBPF | Software stack

- The BPF Compiler Collection (BCC)
  - https://github.com/iovisor/bcc
  - Frontends in Python and Lua
  - The code is compiled and loaded directly on script execution
  - Includes a set of well tested programs ready for use
- BPFftrace https://github.com/iovisor/bpftrace
  - High-level tracing language inspired by awk
  - Language is limited compared to BCC
- Libbpf
  - https://github.com/libbpf/libbpf
  - **BTF** (BPF Type Format) provides struct information to avoid needing kernel headers
    - It stores information about **all** kernel structures (even those that are not exported in headers)
  - **CO-RE** (BPF Compile-Once Run-Everywhere) allows compiled BPF bytecode to be relocatable, avoiding the need for recompilation by LLVM

eBPF

https://github.com/zoidbergwill/awesome-ebpf