XDP for the Rest of Us

Andy Gospodarek -- Principal Engineer, Broadcom
Jesper Dangaard Brouer -- Principal Engineer, Red Hat

Netdev 2.1, 6-8 April 2017

Motivation for this talk

- XDP is still fairly new (v4.8)
- Discussion often centers around datacenter use-cases, often with expensive processor and NIC hardware
- Benefits of early packet drop (DDoS prevention) are significant on lower-powered systems
- Use some alternative hardware and encourage other driver maintainers to consider adding XDP support

Example Application: XDP DDoS Blacklist

https://github.com/netoptimizer/prototype-kernel

Example Application: XDP DDoS Blacklist

- Non-interactive eBPF program that drops traffic based on:
    - Source IPv4 addresses
    - Destination UDP and/or TCP ports
- Has a command-line tool that queries BPF maps for:
    - Configuration
    - Statistics (real-time and historical)
- Motivated to bring XDP processing to more networks using both x86 and ARM64 CPUs

XDP DDoS Blacklist Configuration

```
# ./xdp_ddos01_blacklist_cmdline --list
{
 "216.239.59.128" : 12024134,
 "64.68.90.1" : 67809613,
 "22" : {
    "TCP" : 0
 },
 "80" : {
    "TCP" : 10,
    "UDP" : 4216804996
 }
}
```

XDP DDoS Blacklist Real-time Stats

```
# ./xdp_ddos01_blacklist_cmdline --stats

XDP_action   pps          pps-human-readable period/sec
XDP_ABORTED  0            0                  1.000571
XDP_DROP     5975205      5,975,205          1.000568
XDP_PASS     1102227      1,102,227          1.000568
XDP_TX       0            0                  1.000568
```

Getting Started with eBPF and XDP

Quick Review: eBPF Fundamentals

- Berkley Packet Filter (BPF) is a special-purpose virtual machine for filtering packets (circa 1992)
- eBPF is a ‘universal in-kernel virtual machine’ that is not necessarily network-specific (circa 2014)
- Programs are small and have programming restrictions
- Primary interaction with outside world is generic key/value store called an eBPF map
- Over a dozen different types of maps and growing

Quick Review: eXpress Data Path (XDP)

- Programmable, high performance networking data-path
- The "packet-page" idea from previous Netdev/Netconf
- Operates on raw packet data (before SKB is allocated) directly in driver
- eBPF program result:
    - DROP (never to be seen again)
    - TX (with or without modification/encapsulation)
    - PASS (on to kernel stack for processing!)

Getting Started with XDP (kernel)

- eBPF added in 3.15, XDP core in 4.8
- Native JIT compiling (higher performance execution): x86_64 (3.16), ARM (3.18), s390 (4.1), PowerPC64 (4.8)
- Mellanox: mlx4 (4.8) and mlx5 (4.9)
- QLogic/Cavium: qede (4.10)
- Virtio_net: (4.10)
- Netronome: nfp (4.10)
- Broadcom: bnxt_en (4.11)

Getting Started with XDP (compilers and distros)

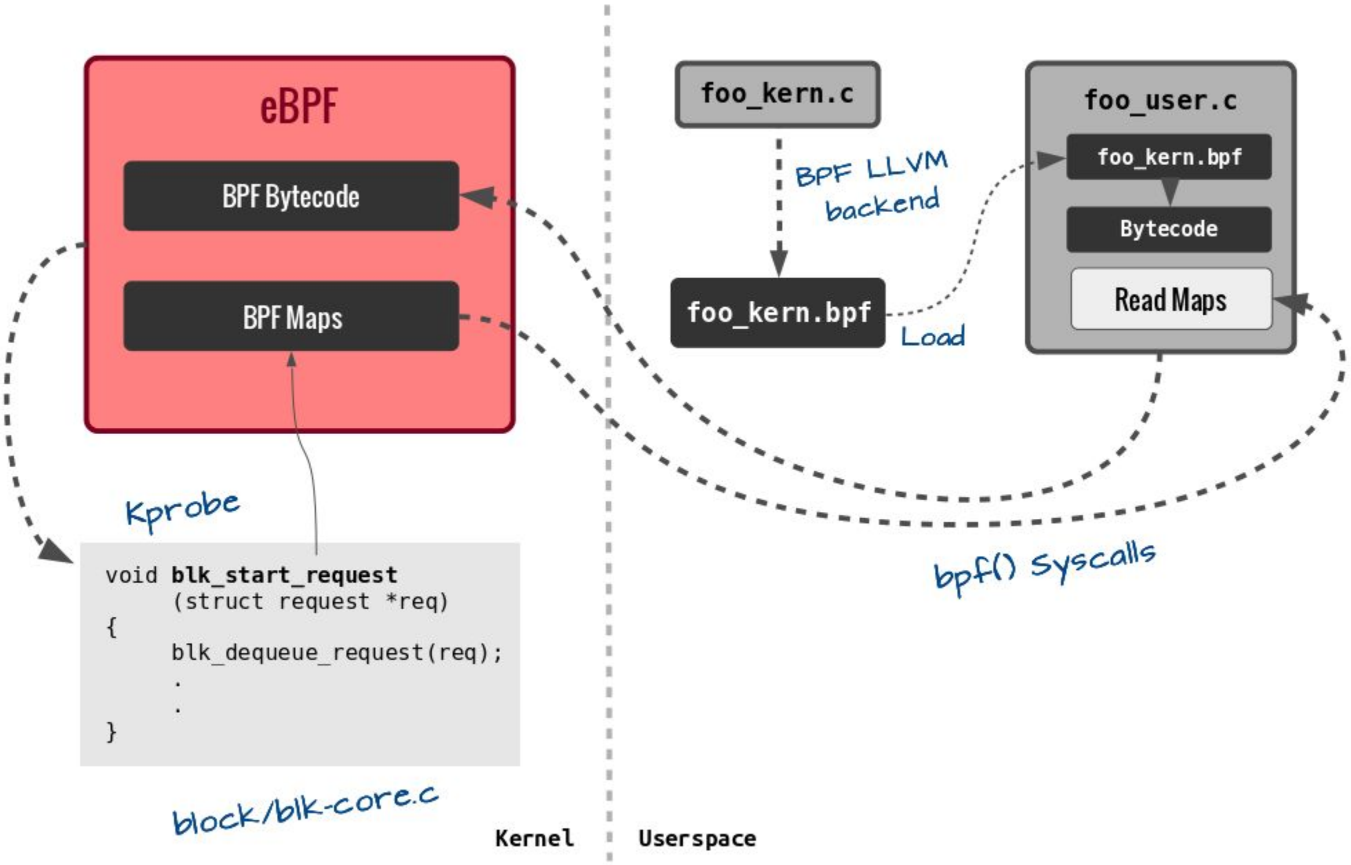
- Roll your own compiler/tools with eBPF support:
  - LLVM >= 3.7.1
  - clang >= 3.4.0
- Fedora25 works with latest kernel and default LLVM/clang
- Ubuntu may require that you load a new kernel and build LLVM/clang yourself for latest features (depending on when you read these slides)

Getting Started with XDP (free samples)

- Three important programs in samples/bpf in kernel tree:
  - xdp1 - drops all frames received on an interface
  - xdp2 - swaps src and dest MAC addresses and retransmits frames on same interface
  - xdp_tx_iptunnel - matches IP and dest port and encapsulates matching frames inside a new IP tunnel and sends them out the same interface

Getting Started with XDP (free samples continued)

- Each program divides source into 2 files:
- eBPF code and map definitions in foo_kern.c
- Output ELF file with map layout and eBPF object code compiled to foo_kern.o by llvm with help from clang
- Userspace code used to load eBPF program and interact with maps compiled to foo_user.o with gcc



```
void blk_start_request
     (struct request *req)
{
     blk_dequeue_request(req);
     .
     .
}
```

From: https://github.com/iovisor/bpf-docs/blob/master/bpf-internals-2.md

Enough already! Let’s see how this application actually works.

File split similar to samples/bpf in kernel source

- **xdp_ddos01_blacklist_kern.c** - map definitions and restricted-C that is compiled into eBPF
- **xdp_ddos01_blacklist_user.c** - userspace code that loads xdp_ddos01_blacklist_kern.o, sets up sysfs links, and exits
- **xdp_ddos01_blacklist_common.h** - a few definitions and blacklist modify operations
- **xdp_ddos01_blacklist_cmdline.c** - code that is compiled into the userspace program used to configure blacklists

The following code has been modified from its original version. It has been formatted to fit this screen, to run in the presentation time allotted, edited for debug statements, code comments, and [possibly] bugs.

xdp_ddos01_blacklist_kern.c

Create per CPU Hash Map

```
struct bpf_map_def SEC("maps") blacklist = {
        .type        = BPF_MAP_TYPE_PERCPU_HASH,
        .key_size    = sizeof(u32),
        .value_size  = sizeof(u64), /* Drop counter */
        .max_entries = 100000,
        .map_flags   = BPF_F_NO_PREALLOC,
};
```

Create Function that Kernel Will Call

```
SEC("xdp_prog")
int xdp_program(struct xdp_md *ctx)
{
        void *data_end = (void *)(long)ctx->data_end;
        void *data     = (void *)(long)ctx->data;
        struct ethhdr *eth = data;
        u16 eth_proto, l3_offset = 0;
        u32 action;

        if (!(parse_eth(eth, data_end, &eth_proto, &l3_offset)))
                bpf_debug("Cannot parse L2: L3off:%llu proto ...
                          l3_offset, eth_proto);
                return XDP_PASS;
        }
        action = handle_eth_protocol(ctx, eth_proto, l3_offset);
        stats_action_verdict(action);
        return action;
}
```

Start Parsing Packets

```
static __always_inline
bool parse_eth(struct ethhdr *eth, void *data_end,
               u16 *eth_proto, u64 *l3_offset)
{
       u16 eth_type;
       u64 offset;

       offset = sizeof(*eth);
       if ((void *)eth + offset > data_end)
               return false;

       eth_type = eth->h_proto;

```

Continue Parsing Packets

```
static __always_inline
u32 handle_eth_protocol(struct xdp_md *ctx, u16 eth_proto, ...
{
        switch (eth_proto) {
        case ETH_P_IP:
                return parse_ipv4(ctx, l3_offset);
                break;
        case ETH_P_IPV6: /* Not handler for IPv6 yet*/
        case ETH_P_ARP:  /* Let OS handle ARP */
                break;
        default:
                bpf_debug("Not handling eth_proto:0x%x\n", eth_proto);
                return XDP_PASS;
        }
        return XDP_PASS;
}
```

Parse IPv4 address and Lookup in Map

```
static __always_inline
u32 parse_ipv4(struct xdp_md *ctx, u64 l3_offset)
{
       void *data_end = (void *)(long)ctx->data_end;
       void *data     = (void *)(long)ctx->data;
       struct iphdr *iph = data + l3_offset;
       u64 *value;
       u32 ip_src; /* type need to match map */

       if (iph + 1 > data_end)
               return XDP_ABORTED;
       ip_src = iph->saddr;

       value = bpf_map_lookup_elem(&blacklist, &ip_src);
       if (value) {
               *value += 1; /* Keep a counter for drop matches */
               return XDP_DROP;
       }
```

Review xdp_ddos01_blacklist_kern.c functionality

- Define map for IPv4 blacklist
- Provide function written in restricted-C that will be called by kernel
- Track whether packet was acceptable or dropped which entry caused drop
- Report decision back to caller
- But how is this code loaded and attached to a netdev?

xdp_ddos01_blacklist_user.c

The Magic of Interacting with eBPF Maps

- BPF library in tools/lib/bpf of kernel tree is extremely helpful for userspace applications, but feels bit magical until you dig into it:
  - `prog_fd[]` entries are populated with each call to `load_bpf_file()`
  - `map_fd[]` entries align with maps declared in program loaded via `load_bpf_file()` (e.g. `foo_kern.o`)

The importance of prog_fd[]

- File descriptors for a BPF program are loaded with **`load_bpf_file(foo_kern.o)`**, stored in **`prog_fd[]`**, and attached to netdev with call to **`set_link_xdp_fd()`**

```
        if (!prog_fd[0]) {
                printf("load_bpf_file: %s\n", strerror(errno));
                return 1;
        }

        if (set_link_xdp_fd(ifindex, prog_fd[0]) < 0) {
                printf("link set xdp fd failed\n");
                return EXIT_FAIL_XDP;
        }
```

The importance of map_fd[]

- File descriptors for each BPF map are loaded with **`load_bpf_file(foo_kern.o)`**, and stored in **`map_fd[]`**, array. Those fds are used as first argument to operate on the map with library calls (from **`tools/lib/bpf/bpf.h`**)

```
int bpf_map_update_elem(int fd, const void *key, const void *value,
                        __u64 flags);

int bpf_map_lookup_elem(int fd, const void *key, void *value);
int bpf_map_delete_elem(int fd, const void *key);
int bpf_map_get_next_key(int fd, const void *key, void *next_key);
int bpf_obj_pin(int fd, const char *pathname);
```

Simple coding for an interactive program (`xdp1_user.c`)

```
	if (load_bpf_file(filename)) {
		printf("%s", bpf_log_buf);
		return 1;
	}

	if (!prog_fd[0]) {
		printf("load_bpf_file: %s\n", strerror(errno));
		return 1;
	}

	signal(SIGINT, int_exit);

	if (set_link_xdp_fd(ifindex, prog_fd[0]) < 0) {
		printf("link set xdp fd failed\n");
		return 1;
	}

	poll_stats(2);
```

Slightly Different Implementation in this Case

- Skipped calling `load_bpf_file()` directly
- `load_bpf_file_fd()` was split into `load_bpf_elf_sections()` and `load_bpf_relocate_maps_and_attach()`
- Between two new functions called `bpf_obj_pin()` to create mapping between each sysfs file and fd for BPF map
- Allows access to eBPF maps by other processes via `bpf_obj_get()`

Upon xdp_ddos01_blacklist_user exit

- All eBPF maps outlined in `xdp_ddos01_blacklist_kern.c` are pinned to files in `/sys/fs/bpf`
- Valid file descriptors for operating on those eBPF maps can be obtained by calling `bpf_obj_get()`
- eBPF program is attached to netdev and running!

xdp_ddos01_blacklist_cmdline.c
(and xdp_ddos01_blacklist_common.h)

Command Line Tool Operations

- Print blacklist entries and historical stats
- Add/Delete IPv4 address from blacklist
- Add/Delete UDP and TCPs port from blacklist
- Print real-time XDP verdict stats

Read Blacklist Entries

```
fd_blacklist = open_bpf_map(file_blacklist);
blacklist_list_all_ipv4(fd_blacklist);
close(fd_blacklist);
```

Get file descriptor for map file

```c
int open_bpf_map(const char *file)
{
       int fd;

       fd = bpf_obj_get(file);
       if (fd < 0) {
               printf("ERR: Failed to open bpf map file:%s err(%d):%s\n",
                      file, errno, strerror(errno));
               exit(EXIT_FAIL_MAP_FILE);
       }
       return fd;
}
```

Find Each Key in the Hash

```
static void blacklist_list_all_ipv4(int fd)
{
        __u32 key = 0, next_key;
        __u64 value;

        while (bpf_map_get_next_key(fd, &key, &next_key) == 0) {
                printf("%s", key ? "," : "" );
                key = next_key;
                value = get_key32_value64_percpu(fd, key);
                blacklist_print_ipv4(key, value);
        }
        printf("%s", key ? "," : "");
}
```

Lookup 32-bit key and Return 64-bit per CPU Sum

```
static __u64 get_key32_value64_percpu(int fd, __u32 key)
{
        unsigned int nr_cpus = bpf_num_possible_cpus();
        __u64 values[nr_cpus];
        __u64 sum = 0;
        int i;

        if ((bpf_map_lookup_elem(fd, &key, values)) != 0) {
                fprintf(stderr,
                        "ERR: bpf_map_lookup_elem failed key:0x%X\n", key);
                return 0;
        }
        for (i = 0; i < nr_cpus; i++) {
                sum += values[i];
        }
        return sum;
}
```

Print IPv4 address drop count

```c
static void blacklist_print_ipv4(__u32 ip, __u64 count)
{
        char ip_txt[INET_ADDRSTRLEN] = {0};

        /* Convert IPv4 addresses from binary to text form */
        if (!inet_ntop(AF_INET, &ip, ip_txt, sizeof(ip_txt))) {
                fprintf(stderr,
                        "ERR: Cannot convert u32 IP:0x%X to IP-txt\n", ip);
                exit(EXIT_FAIL_IP);
        }
        printf("\n \"%s\" : %llu", ip_txt, count);
}
```

Modify Blacklist

```
fd_blacklist = open_bpf_map(file_blacklist);
res = blacklist_modify(fd_blacklist, ip_string, action);
close(fd_blacklist);
```

Add New IPv4 Address to Blacklist

```
static int blacklist_modify(int fd, char *ip_string, uint action)
{
        unsigned int nr_cpus = bpf_num_possible_cpus();
        __u64 values[nr_cpus];
        __u32 key;
        int res;

        memset(values, 0, sizeof(__u64) * nr_cpus);

        /* Convert IP-string into 32-bit network byte-order value */
[...]
        if (action == ACTION_ADD) {
                res = bpf_map_update_elem(fd, &key, values, BPF_NOEXIST);
        } else if (action == ACTION_DEL) {
                res = bpf_map_delete_elem(fd, &key);
        } else {
                return EXIT_FAIL_OPTION;
        }
```

XDP DDoS Blacklist Usage

https://github.com/netoptimizer/prototype-kernel

Attaching eBPF Program to an Interface

```
# ./xdp_ddos01_blacklist --dev enp1s0f1d1
Documentation:
 XDP: DDoS protection via IPv4 blacklist

This program loads the XDP eBPF program into the kernel.
Use the cmdline tool for add/removing source IPs to the blacklist
and read statistics.

 - Attached to device:enp1s0f1d1 (ifindex:3)
 - Blacklist     map file: /sys/fs/bpf/ddos_blacklist
 - Verdict stats map file: /sys/fs/bpf/ddos_blacklist_stat_verdict
 - Blacklist Port map file: /sys/fs/bpf/ddos_port_blacklist
 - Verdict port stats map file: /sys/fs/bpf/ddos_port_blacklist_count_tcp
 - Verdict port stats map file: /sys/fs/bpf/ddos_port_blacklist_count_udp
load_bpf_file: Success
# ip link show enp1s0f1d1
3: enp1s0f1d1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 xdp qdisc mq state ...
    link/ether 00:0a:f7:94:ef:2d brd ff:ff:ff:ff:ff:ff
```

Blacklist configuration

```
# ./xdp_ddos01_blacklist_cmdline --add --ip 10.10.10.10
blacklist_modify() IP:10.10.10.10 key:0xA0A0A0A
# ./xdp_ddos01_blacklist_cmdline --list
{
 "10.10.10.10" : 0,
}
# ./xdp_ddos01_blacklist_cmdline --del --ip 10.10.10.10
blacklist_modify() IP:10.10.10.10 key:0xA0A0A0A
# ./xdp_ddos01_blacklist_cmdline --list
{
}
```

Automating Blacklist Configuration

- Fail2ban relies on external programs to block real hosts who attempt to access a system in a variety of ways
- Extremely easy to write a new module to call `xdp_ddos_blacklist` to initialize XDP and `xdp_ddos01_blacklist_cmdline` to add a host to blacklist
- Change needed committed here: https://github.com/gospo/fail2ban

XDP DDos Blacklist Performance

The Real Reason to use XDP

- CPU: Intel i7-6700K CPU @ 4.00GHz
- NIC: 50GbE Mellanox-CX4 (driver: mlx5)
- Single Stream UDP traffic from remote host
- Delivery to closed UDP port (after commit 9f2f27a9):
  - UdpNoPorts 3,143,931 pps (no iptables/netfilter)

-
  ```
  iptables -t raw -I PREROUTING -p udp --dport 9 -j DROP
  ```
  - Drop: 4,522,956 pps
- XDP blacklist:
  - Drop: 9,697,564 pps

Can the DDoS Blacklist program protect us?

Simulate attack using pktgen_sample05_flow_per_thread.sh with 8 threads with 7 flows using blacklisted IPv4 addresses.

```
$ ./xdp_ddos01_blacklist_cmdline --stats

XDP_action   pps        pps-human-readable period/sec
XDP_ABORTED  0          0                  1.000089
XDP_DROP     30463237   30,463,237         1.000089
XDP_PASS     3438094    3,438,094          1.000089
XDP_TX       0          0                  1.000089
```

Another Reason to use XDP

- CPU: Broadcom BCM5871x (Quad-core A57) @ 1.8GHz
- NIC 10GbE Embedded NIC (driver: bnxt_en)
- Single Stream UDP traffic from remote host
- Delivery to closed UDP port (after commit 9f2f27a9):
  - UdpNoPorts 666,823 pps (no iptables/netfilter)
- XDP blacklist:
  - Drop: 3,721,040 pps

Tips and Tricks
(Problems I encountered)

Turn on in-kernel JIT

- If `perf top` or `perf report` indicate `__bpf_prog_run()` as the top consumer of cycles, check that `bpf_jit_enable=1` is set.

```
$ sysctl net/core/bpf_jit_enable=1
net.core.bpf_jit_enable = 1
```

- Reload all BPF/XDP programs that were running to have this sysctl setting take effect
- On both x86 and ARM64, close to double number of pps handled when in-kernel JIT is enabled

ulimit Settings

- The eBPF maps uses locked memory, which is default very low. Your program likely need to increase resource limit RLIMIT_MEMLOCK see system call setrlimit(2) or run `ulimit -a` to see how to increase shell defaults.
- The bpf_create_map call will return errno EPERM (Operation not permitted) when the RLIMIT_MEMLOCK memory size limit is exceeded.

Dumping maps with readelf or llvm-objdump

```
$ llvm-objdump -h xdp_ddos01_blacklist_kern.o

xdp_ddos01_blacklist_kern.o:  file format ELF64-BPF

Sections:
Idx Name          Size      Address          Type
  0               00000000 0000000000000000
  1 .strtab       0000011d 0000000000000000
  2 .text         000001a8 0000000000000000 TEXT DATA
  3 .rel.text     00000030 0000000000000000
  4 xdp_prog      00000360 0000000000000000 TEXT DATA
  5 .relxdp_prog  00000050 0000000000000000
  6 maps          00000064 0000000000000000 DATA
  7 license       00000004 0000000000000000 DATA
  8 .eh_frame     00000040 0000000000000000 DATA
  9 .rel.eh_frame 00000020 0000000000000000
 10 .symtab       00000258 0000000000000000
```

printk debugging

```
#ifdef  DEBUG
/* Only use this for debug output. Notice output from bpf_trace_printk()
 * end-up in /sys/kernel/debug/tracing/trace_pipe
 */
#define bpf_debug(fmt, ...)                                             \
               ({                                                      \
                       char ____fmt[] = fmt;                           \
                       bpf_trace_printk(____fmt, sizeof(____fmt),      \
                                    ##__VA_ARGS__);                    \
               })
#else
#define bpf_debug(fmt, ...) { } while (0)
#endif
```

Possibly Useful Hacks^W Ideas

Whitelisting

```
# basically...
$ sed -i s/XDP_DROP/XDP_SAVEDROP/g *.[ch]
$ sed -i s/XDP_PASS/XDP_DROP/g *.[ch]
$ sed -i s/XDP_SAVEDROP/XDP_PASS/g *.[ch]
```

Blacklisting subnets

- LPM map type (BPF_MAP_TYPE_LPM_TRIE) already implemented by Daniel Mack!
- Add new BPF map (xdp_ddos01_blacklist_kern.c)
- Add code in to lookup LPM entries for newly created map in function parse_ipv4() (xdp_ddos01_blacklist_kern.c)
- Add new command line options and JSON output (xdp_ddos01_blacklist_cmdline.c)

IPv6 blacklisting

- New PERCPU Hash long enough to hold IPv6 addresses (xdp_ddos01_blacklist_kern.c)
- Create new function to check against new array similar to parse_ipv4() and call from handle_eth_protocol() (xdp_ddos01_blacklist_kern.c)
- Add new command line options and JSON output (xdp_ddos01_blacklist_cmdline.c)

What’s next for XDP DDoS Blacklist

Keeping Maps During eBPF Program Attach/Detach

- Possible as long as application schema does not change in an incompatible manner
- Work from Daniel Borkmann in iproute2 codebase detects differences between map types and could be used to keep existing maps that are the same type across reloads

--import Option to xdp_ddos01_blacklist_cmdline

- Parallel to the --list option
- Import/merge based on JSON output from previous run or from remote system

Add support for custom processing pipeline

- Add a **`BPF_MAP_TYPE_PROG_ARRAY`** map that can be traversed with **`bpf_tail_call()`**
- Allow users to more easily create custom processing pipeline (i.e. use only IPv4 address filtering rather than checking addresses and TCP/UDP ports)

References

- https://github.com/iovisor/bpf-docs/blob/master/Express_Data_Path.pdf
- https://prototype-kernel.readthedocs.io/en/latest/
- https://github.com/iovisor/bcc/blob/master/docs/kernel-versions.md
- https://www.slideshare.net/AlexeiStarovoitov/bpf-inkernel-virtual-machine
- https://www.slideshare.net/brendangregg/linux-bpf-superpowers
- man bpf
- http://www.tcpdump.org/papers/bpf-usenix93.pdf
- https://github.com/iovisor/bpf-docs/blob/master/bpf-internals-2.md
- http://people.netfilter.org/hawk/presentations/OpenSourceDays2017/XDP_DDoS_protecting_osd2017.pdf