

Efficient System Monitoring in Cloud Native Environments

gergely.szabo@origoss.com

About Myself

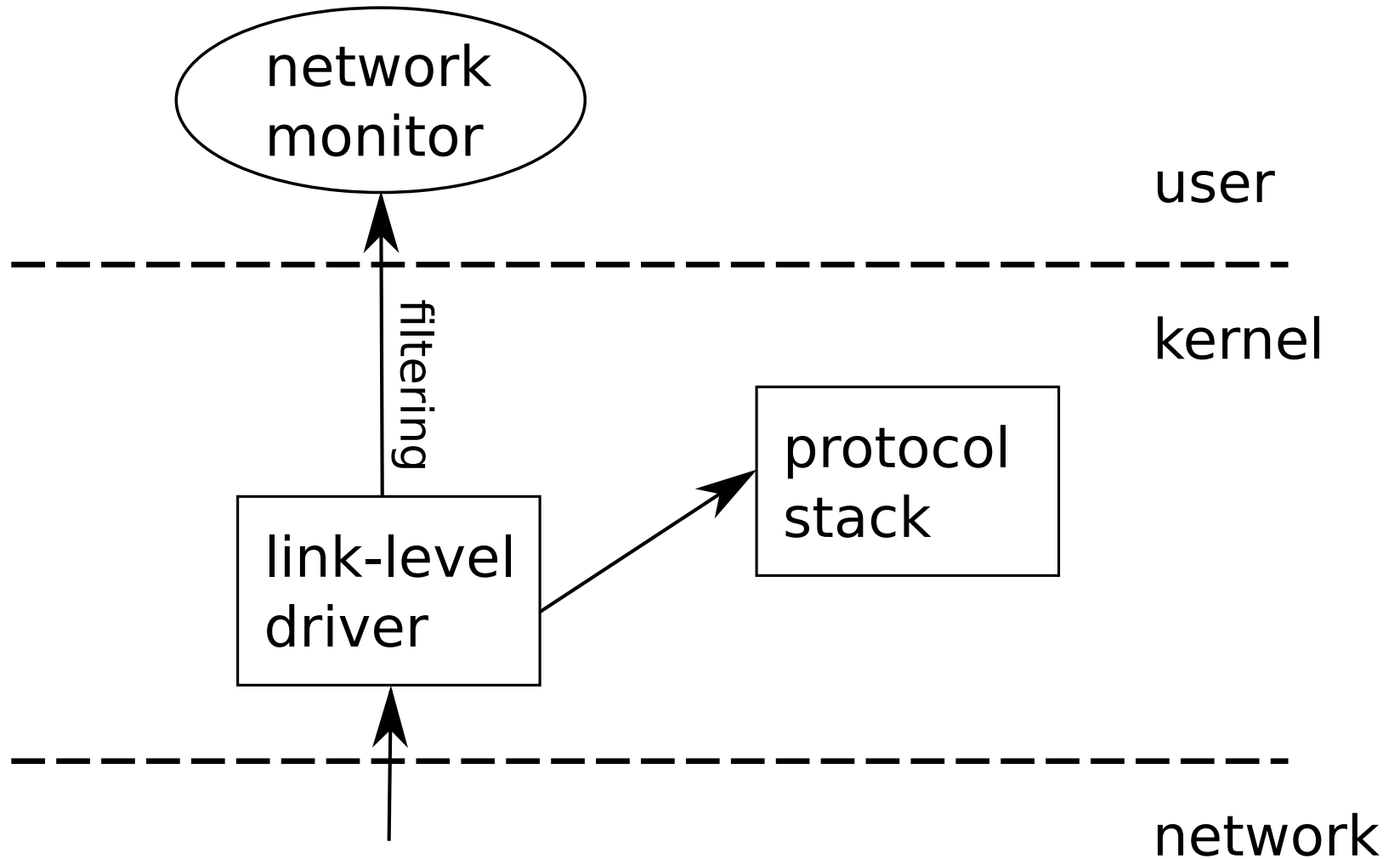
- more than 15 years in the industry
- research, development, system architect, etc...
- currently at Origoss Solutions
 - Cloud Native
 - Kubernetes
 - Prometheus

Agenda

- BPF
- Linux kernel tracing
- EBPF
- EBPF-based monitoring in the cloud

BPF

Packet Filtering Problem



Filtering Requirements

- Efficient
- Flexible filter rules
- Safe

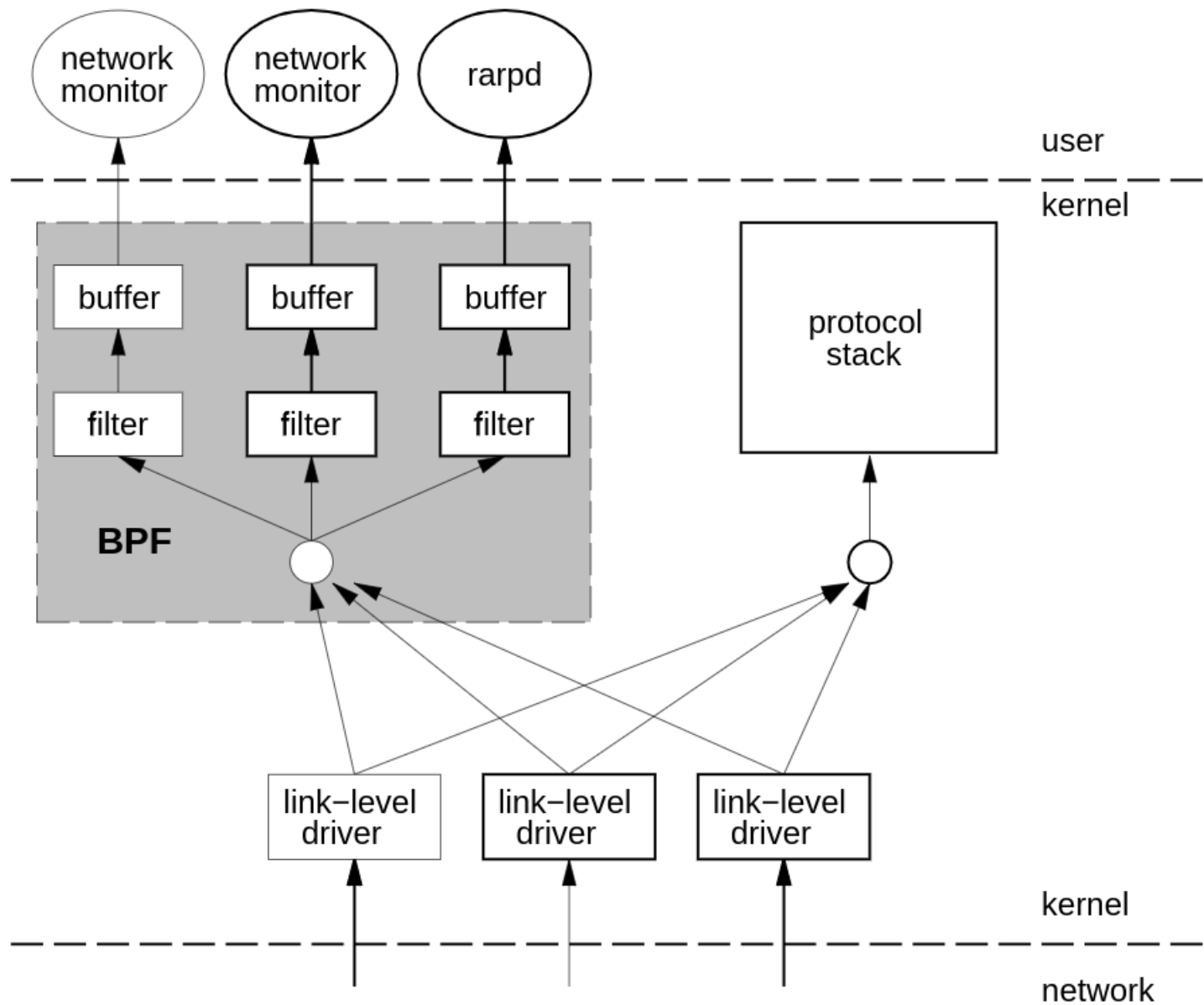
BPF

Steven McCanne and Van Jacobson:

**The BSD Packet Filter: A New Architecture for User-level
Packet Capture, 1992**

<http://www.tcpdump.org/papers/bpf-usenix93.pdf>
(<http://www.tcpdump.org/papers/bpf-usenix93.pdf>).

BPF Architecture



Capturing without Filtering

In []: %%bash
sudo tcpdump -nc 4

Simple Filtering Rule

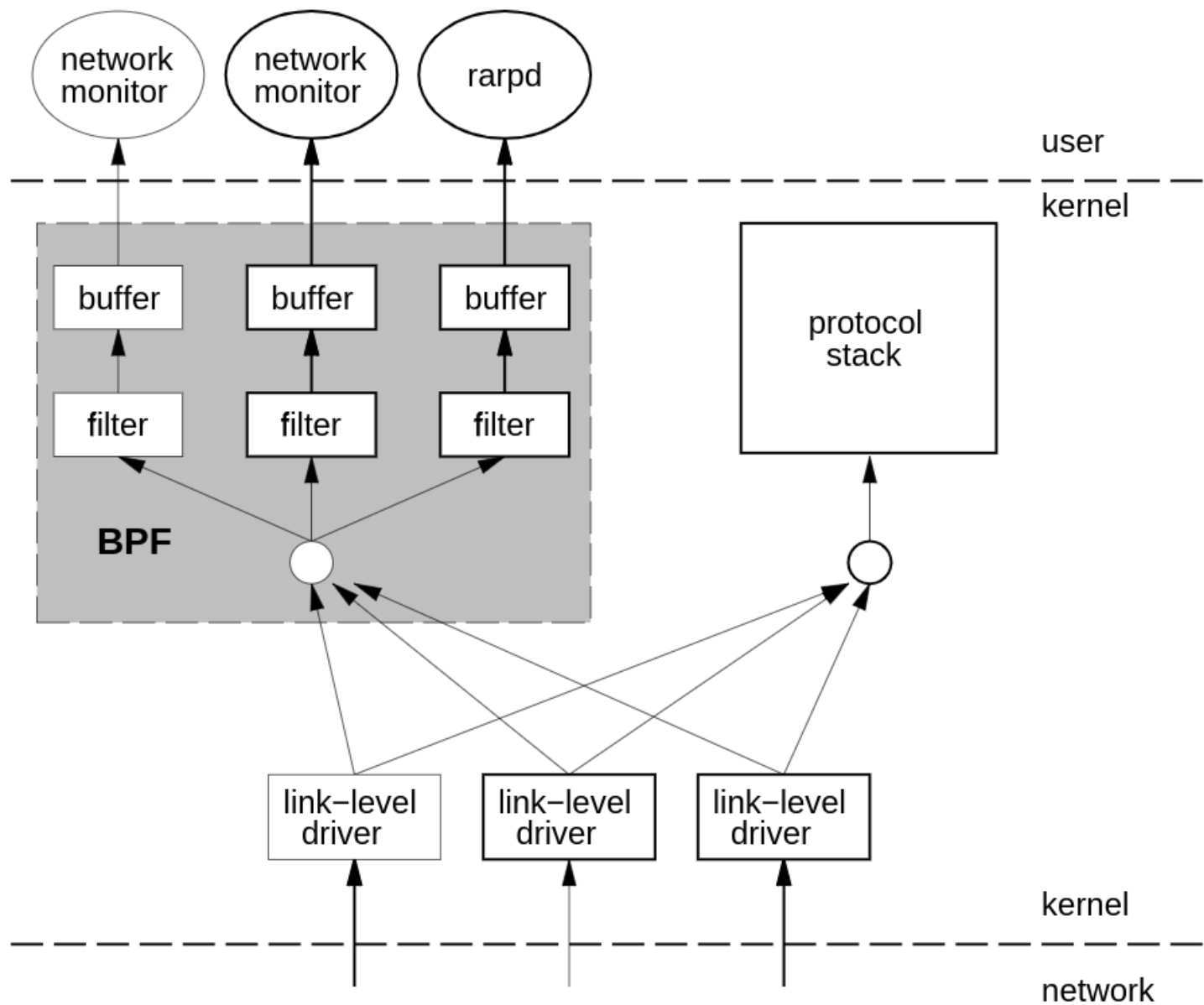
In []: %%bash
sudo tcpdump -nc 4 tcp and port 80

Complex Rule

To print all IPv4 HTTP packets to and from port 80, i.e. print only packets that contain data, not, for example, SYN and FIN packets and ACK-only packets.

```
In [ ]: %%bash
sudo tcpdump -nc 4 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)'
```

How Does This Work?



BPF VM Instruction Set

opcodes		addr modes			
ldb	[k]			[x+k]	
ldh	[k]			[x+k]	
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k			x	
sub	#k			x	
mul	#k			x	
div	#k			x	
and	#k			x	
or	#k			x	
lsh	#k			x	
rsh	#k			x	
ret	#k			a	
tax					
txa					

Simple Filtering Rule

In []: %%bash
tcpdump -d tcp and port 80

Complex Rule

```
In [ ]: %%bash
tcpdump -d 'tcp port 80 and (((ip[2:2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>
2)) != 0)'
```

Linux Kernel Tracepoints

- A tracepoint placed in code provides a hook to call a function (probe) that you can provide at runtime.
- A tracepoint can be "on" or "off"
 - When a tracepoint is "on", the function you provide is called each time the tracepoint is executed
- They can be used for tracing and performance accounting.

Adding Tracepoints

```
void blk_requeue_request(struct request_queue *q, struct request *rq)
{
    blk_delete_timer(rq);
    blk_clear_rq_complete(rq);
    trace_block_rq_requeue(q, rq);    // <- Tracepoint hook

    if (rq->cmd_flags & REQ_QUEUED)
        blk_queue_end_tag(q, rq);

    BUG_ON(blk_queued_rq(rq));

    elv_requeue_request(q, rq);
}
```

List of Tracepoints

In []: %%bash
perf list tracepoint

Tracepoints in Action

```
In [ ]: %%bash  
sudo perf stat -a -e kmem:kmalloc sleep 10
```

Linux Kernel KProbes

- dynamically break into any kernel routine and collect debugging and performance information non-disruptively.
 - some parts of the kernel code can not be trapped
- two types of probes: kprobes, and kretprobes
- A kprobe can be inserted on virtually any instruction in the kernel.
- A return probe fires when a specified function returns.

List of KProbes

```
In [ ]: %%bash  
sudo cat /sys/kernel/debug/kprobes/list
```


Probing a Linux Function

```
void blk_delete_timer(struct request *req)
{
    list_del_init(&req->timeout_list);
}
```

```
In [ ]: %%bash
sudo sh -c 'echo p:demo_probe blk_delete_timer >> /sys/kernel/debug/tracing/kprobe_events'
```

List of KProbes

```
In [ ]: %%bash
sudo cat /sys/kernel/debug/kprobes/list
```

```
In [ ]: %%bash
sudo perf list | grep demo
```

KProbes in Action

```
In [ ]: %%bash  
sudo perf stat -a -e kprobes:demo_probe sleep 10
```

Removing KProbe

```
In [ ]: %%bash
sudo sh -c 'echo "-:demo_probe" >> /sys/kernel/debug/tracing/kprobe_events'
```

```
In [ ]: %%bash
sudo cat /sys/kernel/debug/kprobes/list
```

```
In [ ]: %%bash
sudo perf list | grep demo
```

EBPF

Recent Developments: eBPF

- v3 . 15: BPF machine upgrade (64bit registers, more registers, new instruction)
- v3 . 16: JIT compiling
- v3 . 18: BPF maps
- v4 . 1: attach BPF programs to kprobes
- v4 . 7: attach BPF programs to tracepoints
- v4 . 8: XDP (<https://www.iovisor.org/technology/xdp>).
- ...

eBPF Maps

- 15+ map types: `BPF_MAP_TYPE_HASH`, `BPF_MAP_TYPE_ARRAY`, `BPF_MAP_TYPE_PROG_ARRAY`, `BPF_MAP_TYPE_PERF_EVENT_ARRAY`, ...
- associated to a userspace process
- read/written by userspace process, eBPF programs

eBPF Map Operations

```
int bpf_create_map(enum bpf_map_type map_type, unsigned int key_size, unsigned
    int value_size, unsigned int max_entries)
int bpf_lookup_elem(int fd, const void *key, void *value)
int bpf_update_elem(int fd, const void *key, const void *value, uint64_t flags
)
int bpf_delete_elem(int fd, const void *key)
int bpf_get_next_key(int fd, const void *key, void *next_key)
```


eBPF Programs

- 20+ program types: BPF_PROG_TYPE_SOCKET_FILTER, BPF_PROG_TYPE_KPROBE, BPF_PROG_TYPE_TRACEPOINT, BPF_PROG_TYPE_XDP, ...
- associated to a userspace process
- event-based execution (e.g. tracepoint hooks)
- executed by BPF VM
 - safe
 - efficient

eBPF Program Operations

```
int bpf_prog_load(enum bpf_prog_type type, const struct bpf_insn *insns, int i  
nsn_cnt, const char *license)
```

eBPF Program as C struct

```
struct bpf_insn prog[] = {
    BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),          /* r6 = r1 */
    BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol)),
                                                    /* r0 = ip->proto */
    BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4),
                                                    /* *(u32 *)(fp - 4) = r0 */
    BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),          /* r2 = fp */
    BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4),         /* r2 = r2 - 4 */
    BPF_LD_MAP_FD(BPF_REG_1, map_fd),              /* r1 = map_fd */
    BPF_CALL_FUNC(BPF_FUNC_map_lookup_elem),
                                                    /* r0 = map_lookup(r1, r2) */
    BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
                                                    /* if (r0 == 0) goto pc+2 */
    BPF_MOV64_IMM(BPF_REG_1, 1),                   /* r1 = 1 */
    BPF_XADD(BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0),
                                                    /* lock *(u64 *) r0 += r1 */
    BPF_MOV64_IMM(BPF_REG_0, 0),                   /* r0 = 0 */
    BPF_EXIT_INSN(),                               /* return r0 */
};
```

eBPF Program as C Code

Can be compiled with LLVM/Clang using the BPF backend.

```
int bpf_prog1(struct pt_regs *ctx)
{
    /* attaches to kprobe netif_receive_skb,
     * looks for packets on loopback device and prints them
     */
    char devname[IFNAMSIZ];
    struct net_device *dev;
    struct sk_buff *skb;
    int len;

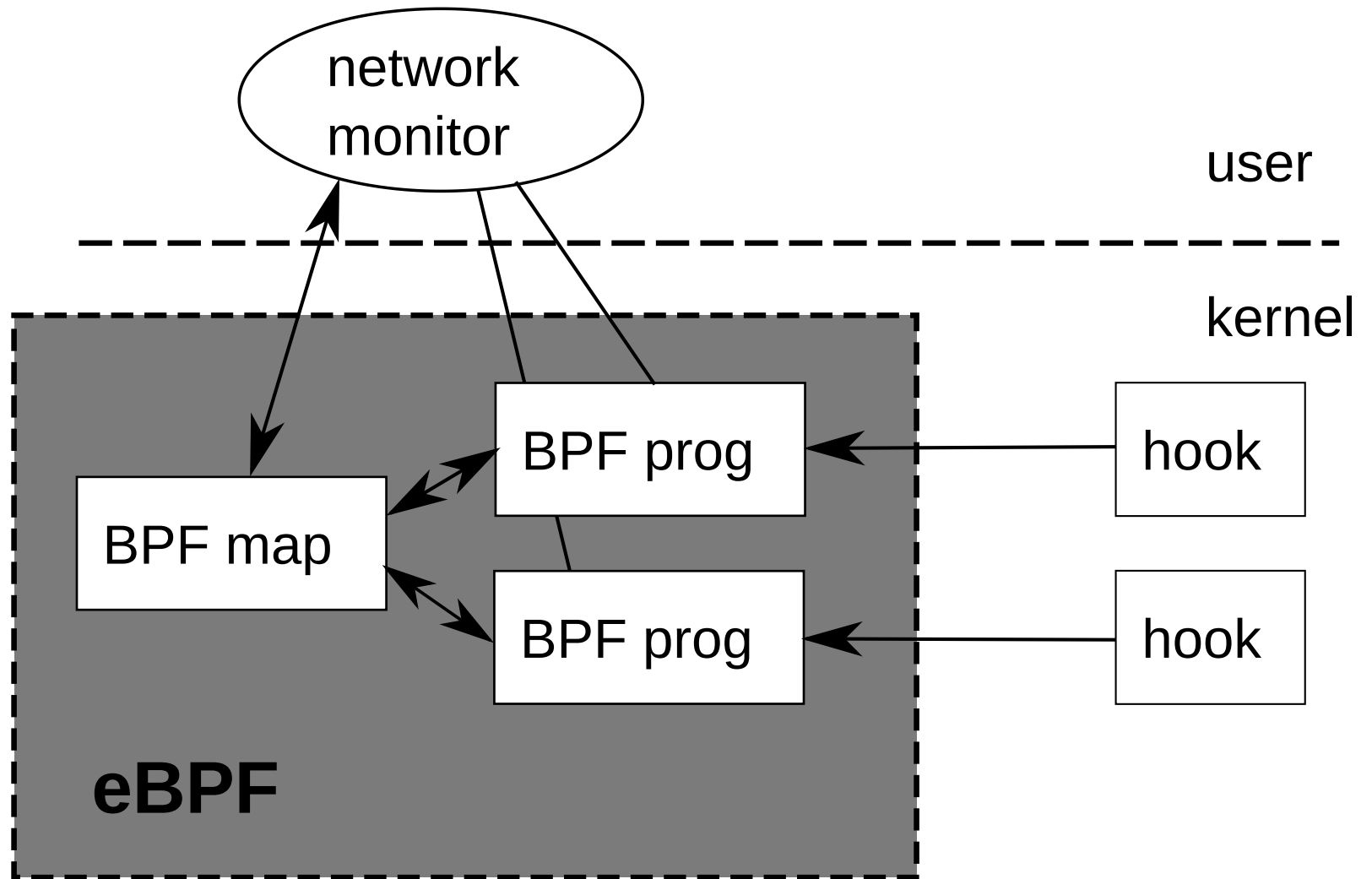
    /* non-portable! works for the given kernel only */
    skb = (struct sk_buff *) PT_REGS_PARM1(ctx);
    dev = _(skb->dev);
    len = _(skb->len);

    bpf_probe_read(devname, sizeof(devname), dev->name);

    if (devname[0] == 'l' && devname[1] == 'o') {
        char fmt[] = "skb %p len %d\n";
        /* using bpf_trace_printk() for DEBUG ONLY */
        bpf_trace_printk(fmt, sizeof(fmt), skb, len);
    }

    return 0;
}
```

eBPF-based Monitoring



eBPF Workflow: Linux Kernel BPF Samples

- see `linux/samples/bpf`
- eBPF kernel part (.c)
 - contains map and program definitions
 - compiled with LLVM -> .o
- eBPF user part (.c)
 - compiles to executable
 - extracts maps and programs from kernel part (.o)
 - creates maps: `bpf_create_map`
 - relocates maps in program codes
 - loads programs: `bpf_prog_load`
 - reads maps and generates output

eBPF Workflow: **iovisor/bcc**

- see <https://github.com/iovisor/bcc> (<https://github.com/iovisor/bcc>).
- single Python script that contains:
 - definition of eBPF maps
 - definition of eBPF programs (as LLVM compatible C code)
 - code to read and process the maps
- C code is compiled when the script starts (LLVM)

eBPF Example

<https://github.com/iovisor/bcc/blob/master/tools/filelife.py>
.<https://github.com/iovisor/bcc/blob/master/tools/filelife.py>).


```

#!/usr/bin/python
# @lint-avoid-python-3-compatibility-imports
#
# filelife      Trace the lifespan of short-lived files.
#                For Linux, uses BCC, eBPF. Embedded C.
#
# This traces the creation and deletion of files, providing information
# on who deleted the file, the file age, and the file name. The intent is to
# provide information on short-lived files, for debugging or performance
# analysis.
#
# USAGE: filelife [-h] [-p PID]
#
# Copyright 2016 Netflix, Inc.
# Licensed under the Apache License, Version 2.0 (the "License")
#
# 08-Feb-2015    Brendan Gregg    Created this.
# 17-Feb-2016    Allan McAleavy   updated for BPF_PERF_OUTPUT

from __future__ import print_function
from bcc import BPF
import argparse
from time import strftime
import ctypes as ct

# arguments
examples = """examples:
    ./filelife          # trace all stat() syscalls
    ./filelife -p 181   # only trace PID 181
"""

parser = argparse.ArgumentParser(
    description="Trace stat() syscalls",
    formatter_class=argparse.RawDescriptionHelpFormatter,
    epilog=examples)
parser.add_argument("-p", "--pid",
    help="trace this PID only")
parser.add_argument("--ebpf", action="store_true",

```

```

help=argparse.SUPPRESS)
args = parser.parse_args()
debug = 0

# define BPF program
bpf_text = """
#include <uapi/linux/ptrace.h>
#include <linux/fs.h>
#include <linux/sched.h>
struct data_t {
    u32 pid;
    u64 delta;
    char comm[TASK_COMM_LEN];
    char fname[DNAME_INLINE_LEN];
};
BPF_HASH(birth, struct dentry *);
BPF_PERF_OUTPUT(events);
// trace file creation time
int trace_create(struct pt_regs *ctx, struct inode *dir, struct dentry *dentry)
{
    u32 pid = bpf_get_current_pid_tgid();
    FILTER
    u64 ts = bpf_ktime_get_ns();
    birth.update(&dentry, &ts);
    return 0;
};
// trace file deletion and output details
int trace_unlink(struct pt_regs *ctx, struct inode *dir, struct dentry *dentry)
{
    struct data_t data = {};
    u32 pid = bpf_get_current_pid_tgid();
    FILTER
    u64 *tsp, delta;
    tsp = birth.lookup(&dentry);
    if (tsp == 0) {
        return 0;    // missed create
    }
}

```

```

    }
    delta = (bpf_ktime_get_ns() - *tsp) / 1000000;
    birth.delete(&dentry);
    struct qstr d_name = dentry->d_name;
    if (d_name.len == 0)
        return 0;
    if (bpf_get_current_comm(&data.comm, sizeof(data.comm)) == 0) {
        data.pid = pid;
        data.delta = delta;
        bpf_probe_read(&data.fname, sizeof(data.fname), d_name.name);
    }
    events.perf_submit(ctx, &data, sizeof(data));
    return 0;
}
"""

```

```

TASK_COMM_LEN = 16          # linux/sched.h
DNAME_INLINE_LEN = 255     # linux/dcache.h

```

```

class Data(ct.Structure):
    _fields_ = [
        ("pid", ct.c_uint),
        ("delta", ct.c_ulonglong),
        ("comm", ct.c_char * TASK_COMM_LEN),
        ("fname", ct.c_char * DNAME_INLINE_LEN)
    ]

if args.pid:
    bpf_text = bpf_text.replace('FILTER',
        'if (pid != %s) { return 0; }' % args.pid)
else:
    bpf_text = bpf_text.replace('FILTER', '')
if debug or args.ebpf:
    print(bpf_text)
    if args.ebpf:
        exit()

```

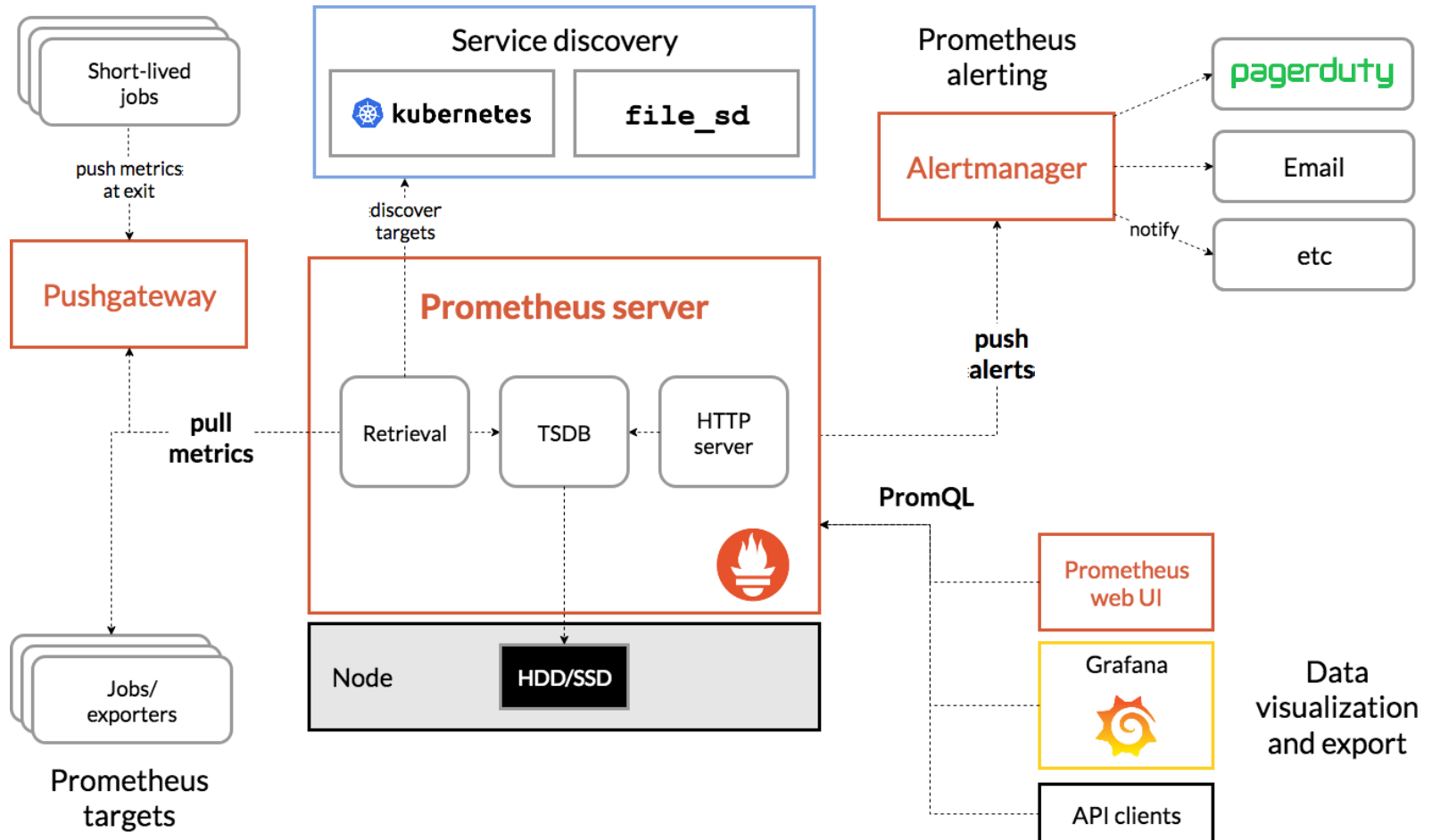
```

# initialize BPF

```

Prometheus

<https://prometheus.io/docs/introduction/overview/#architecture>
(<https://prometheus.io/docs/introduction/overview/#architecture>)



Prometheus eBPF Exporter

Motivation of this exporter is to allow you to write eBPF code and export metrics that are not otherwise accessible from the Linux kernel.

- https://github.com/cloudflare/ebpf_exporter
(https://github.com/cloudflare/ebpf_exporter).
- leverages on tools made by `iovisor/bcc`
- written in Go
- exported metrics are defined as yaml files

Prometheus eBPF Exporter - Example

https://github.com/cloudflare/ebpf_exporter/blob/master/examples/bio.yaml
(https://github.com/cloudflare/ebpf_exporter/blob/master/examples/bio.yaml).

```
programs:
  # See:
  # * https://github.com/iovisor/bcc/blob/master/tools/biolatency.py
  # * https://github.com/iovisor/bcc/blob/master/tools/biolatency\_example.txt
  #
  # See also: bio-tracepoints.yaml
  - name: bio
    metrics:
      histograms:
        - name: bio_latency_seconds
          help: Block IO latency histogram
          table: io_latency
          bucket_type: exp2
          bucket_min: 0
          bucket_max: 26
          bucket_multiplier: 0.000001 # microseconds to seconds
          labels:
            - name: device
              size: 32
              decoders:
                - name: string
            - name: operation
              size: 8
              decoders:
                - name: uint
                - name: static_map
                  static_map:
                    1: read
                    2: write
            - name: bucket
              size: 8
              decoders:
                - name: uint
        - name: bio_size_bytes
          help: Block IO size histogram with kibibyte buckets
          table: io_size
          bucket_type: exp2
```

```

    bucket_min: 0
    bucket_max: 15
    bucket_multiplier: 1024 # kibibytes to bytes
    labels:
      - name: device
        size: 32
        decoders:
          - name: string
      - name: operation
        size: 8
        decoders:
          - name: uint
          - name: static_map
            static_map:
              1: read
              2: write
      - name: bucket
        size: 8
        decoders:
          - name: uint
kprobes:
  blk_start_request: trace_req_start
  blk_mq_start_request: trace_req_start
  blk_account_io_completion: trace_req_completion
code: |
  #include <linux/blkdev.h>
  #include <linux/blk_types.h>
  typedef struct disk_key {
    char disk[32];
    u8 op;
    u64 slot;
  } disk_key_t;
  // Max number of disks we expect to see on the host
  const u8 max_disks = 255;
  // 27 buckets for latency, max range is 33.6s .. 67.1s
  const u8 max_latency_slot = 26;
  // 16 buckets per disk in kib, max range is 16mib .. 32mib
  const u8 max_size_slot = 15;

```



```

    // Hash to temporarily hold the start time of each bio request, max 10k in
    -flight by default
    BPF_HASH(start, struct request *);
    // Histograms to record latencies
    BPF_HISTOGRAM(io_latency, disk_key_t, (max_latency_slot + 1) * max_disk
s);
    // Histograms to record sizes
    BPF_HISTOGRAM(io_size, disk_key_t, (max_size_slot + 1) * max_disks);
    // Record start time of a request
    int trace_req_start(struct pt_regs *ctx, struct request *req) {
        u64 ts = bpf_ktime_get_ns();
        start.update(&req, &ts);
        return 0;
    }
    // Calculate request duration and store in appropriate histogram bucket
    int trace_req_completion(struct pt_regs *ctx, struct request *req, unsig
ned int bytes) {
        u64 *tsp, delta;
        // Fetch timestamp and calculate delta
        tsp = start.lookup(&req);
        if (tsp == 0) {
            return 0; // missed issue
        }
        // There are write request with zero length on sector zero,
        // which do not seem to be real writes to device.
        if (req->__sector == 0 && req->__data_len == 0) {
            return 0;
        }
        // Disk that received the request
        struct gendisk *disk = req->rq_disk;
        // Delta in nanoseconds
        delta = bpf_ktime_get_ns() - *tsp;
        // Convert to microseconds
        delta /= 1000;
        // Latency histogram key
        u64 latency_slot = bpf_log2l(delta);
        // Cap latency bucket at max value
        if (latency_slot > max_latency_slot) {

```

Using eBPF for Monitoring in Cloud-Native Environment

Cloud Native:

- microservice architecture
- containerized
- orchestrated

Pitfall #1: Dependencies

- bcc
- LLVM
- Linux kernel headers

Pitfall #2: KProbes and Kernel Version

```
static int bpf_prog_load(union bpf_attr *attr)
{
    enum bpf_prog_type type = attr->prog_type;
    struct bpf_prog *prog;
    int err;
    char license[128];
    bool is_gpl;

    if (CHECK_ATTR(BPF_PROG_LOAD))
        return -EINVAL;

    if (attr->prog_flags & ~BPF_F_STRICT_ALIGNMENT)
        return -EINVAL;

    /* copy eBPF program license from user space */
    if (strncpy_from_user(license, u64_to_user_ptr(attr->license),
                        sizeof(license) - 1) < 0)
        return -EFAULT;
    license[sizeof(license) - 1] = 0;

    /* eBPF programs must be GPL compatible to use GPL-ed functions */
    is_gpl = license_is_gpl_compatible(license);

    if (attr->insn_cnt == 0 || attr->insn_cnt > BPF_MAXINSNS)
        return -E2BIG;

    if (type == BPF_PROG_TYPE_KPROBE &&
        attr->kern_version != LINUX_VERSION_CODE)
        return -EINVAL;
    /* ... */
}
```

Pitfall #3: KProbes and Stability

- Kprobe can be created for any kernel function
- Most of the Linux kernel source code is subject to change
 - in-kernel APIs and ABIs are unstable
- Distribution-specific kernel modifications, proprietary kernels

Pitfall #4: Kernel Support

- v4 . 1: attach BPF programs to kprobes (21 June, 2015)
- v4 . 7: attach BPF programs to tracepoints (24 July, 2016)
- RHEL 7.6 (30 October, 2018) has 3 . 10 . 0 - 957

Ongoing Activities

- eBPF-based Prometheus exporter, containerized
- run-time configurable eBPF metrics
- self contained
 - no dep on iovisor/bcc
 - no dep on Linux kernel headers
- supporting the major Linux distributions

Thank you!

Questions?