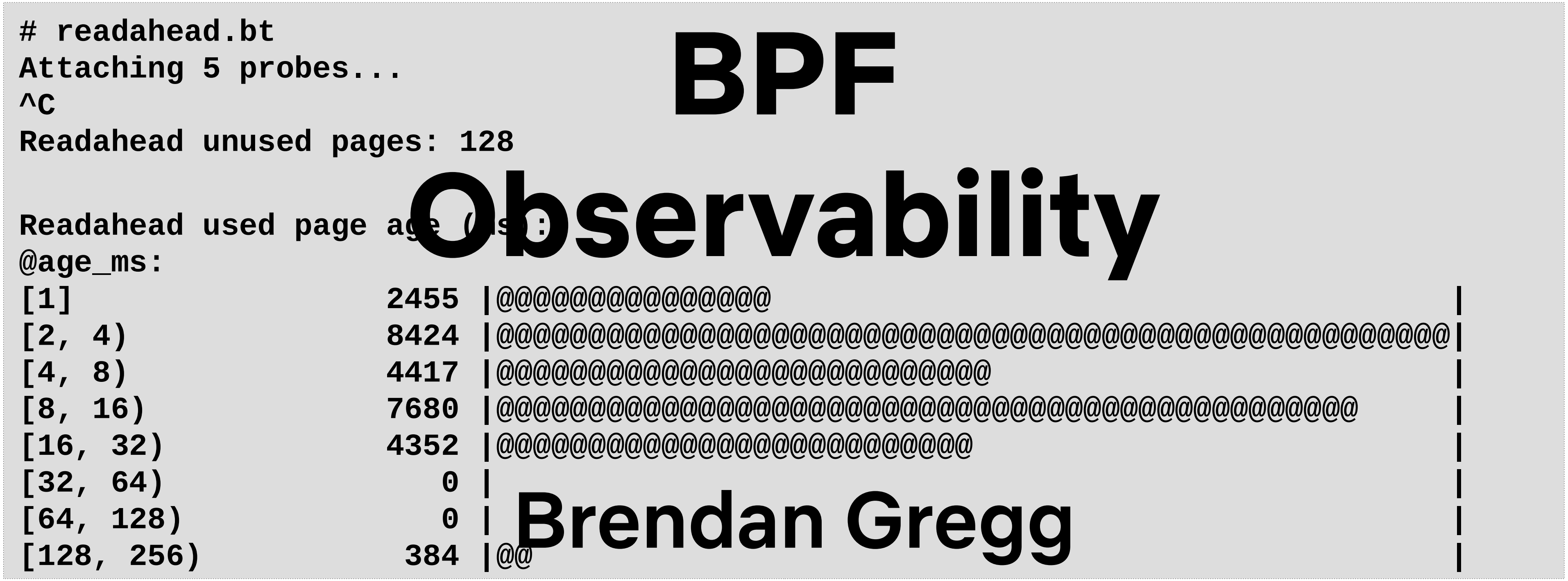


```
# readahead.bt
Attaching 5 probes...
^C
Readahead unused pages: 128

Readahead used page age (ms):
@age_ms:
[1]                 2455 |@@@@@@@@@@@@@@@                                     |
[2, 4)              8424 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)              4417 |@@@@@@@@@@@@@@@@@@@@@@@@@@@                         |
[8, 16)             7680 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
[16, 32)            4352 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)           384 |@@                                                  |
```

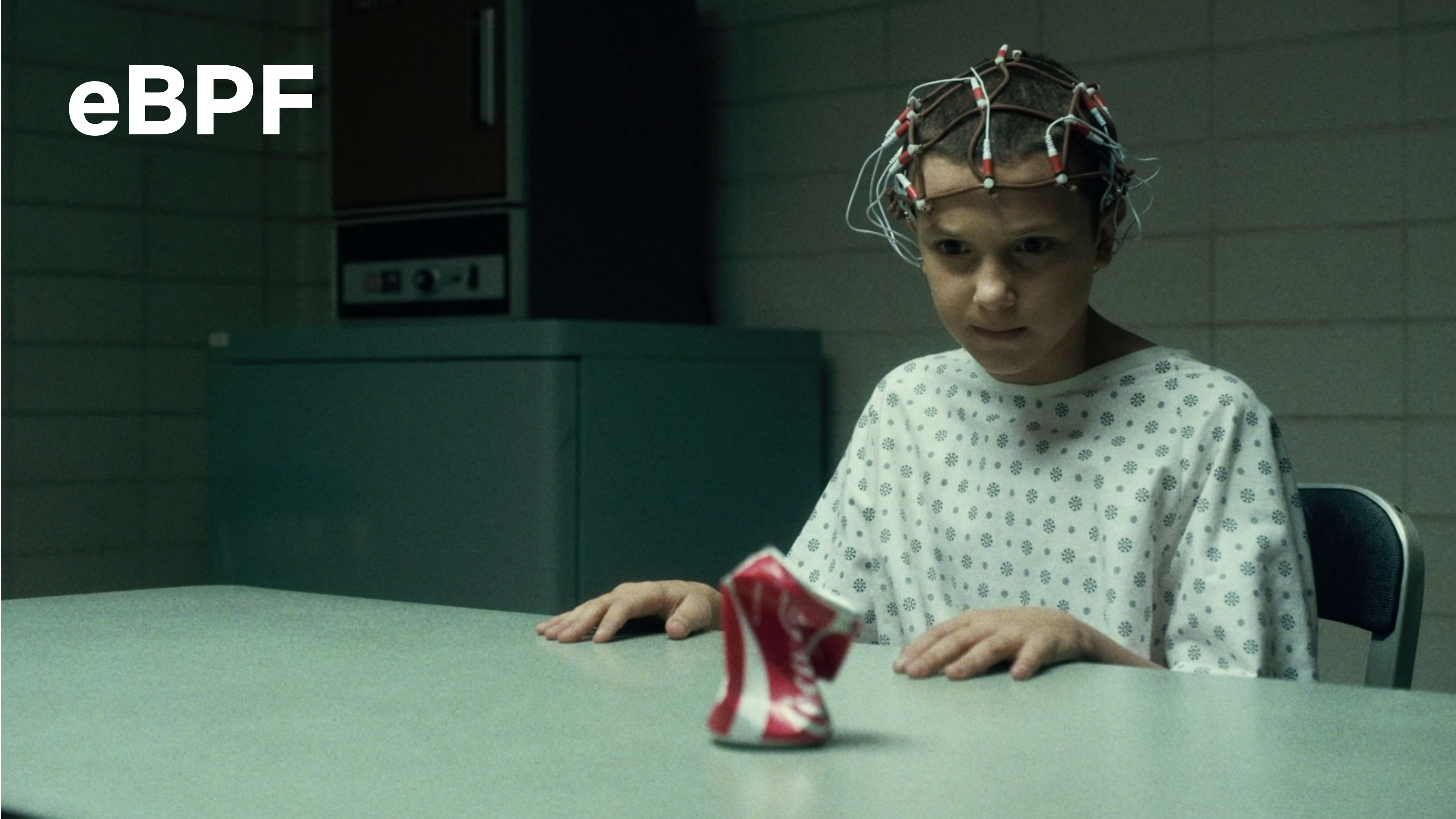
BPF Observability

Brendan Gregg

LSFMM
Apr 2019

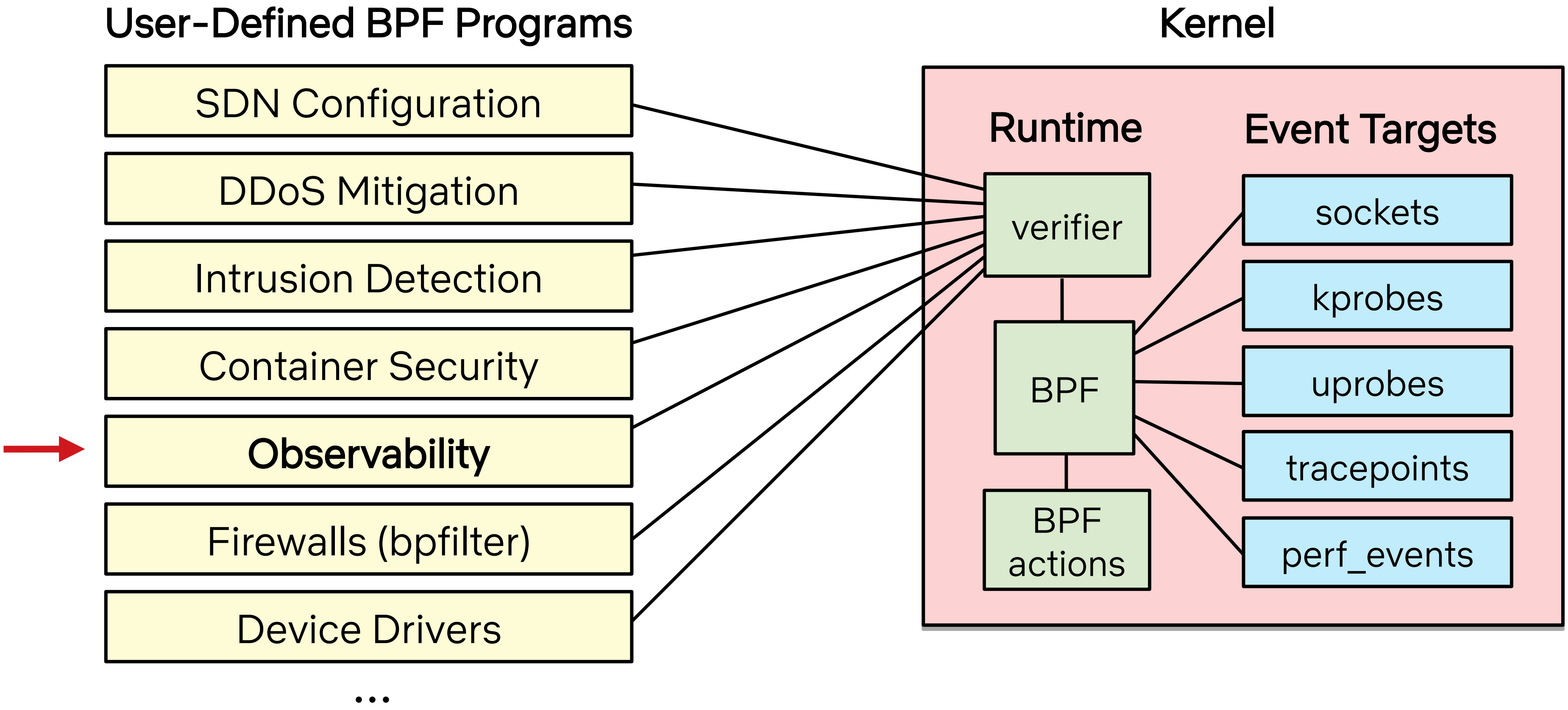
NETFLIX

eBPF



Superpowers Demo

eBPF: extended Berkeley Packet Filter



User-Defined BPF Programs

SDN Configuration

DDoS Mitigation

Intrusion Detection

Container Security

Observability

Firewalls (bpfilter)

Device Drivers

…

Kernel

Runtime

verifier

BPF

BPF actions

Event Targets

sockets

kprobes

uprobes

tracepoints

perf_events

NETFLIX

>150k AWS EC2 server instances

aws

~34% US Internet traffic at night

ubuntu

>130M members

FreeBSD®

Performance is customer satisfaction & Netflix cost

Experience: ps(1) failure

(This is from last week...)

Experience: ps(1) failure

```
# wait for $pid to finish:
while ps -p $pid >/dev/null; do
    sleep 1
done
# do stuff...
```

Experience: ps(1) failure

```
# wait for $pid to finish:
while ps -p $pid >/dev/null; do
    sleep 1
done
# do stuff...
```

Problem: ps(1) sometimes fails to find the process

Hypothesis: kernel bug!

Experience: ps(1) failure

Which syscall is abnormally failing (without strace(1))?

```
# bpftrace -e 't:syscalls:sys_exit_* /comm == "ps"/ {
    @[probe, args->ret > 0 ? 0 : - args->ret] = count(); }'
Attaching 316 probes...
[...]
@[tracepoint:syscalls:sys_exit_openat, 2]: 120
@[tracepoint:syscalls:sys_exit_newfstat, 0]: 180
@[tracepoint:syscalls:sys_exit_mprotect, 0]: 230
@[tracepoint:syscalls:sys_exit_rt_sigaction, 0]: 240
@[tracepoint:syscalls:sys_exit_mmap, 0]: 350
@[tracepoint:syscalls:sys_exit_newstat, 0]: 5000
@[tracepoint:syscalls:sys_exit_read, 0]: 10170
@[tracepoint:syscalls:sys_exit_close, 0]: 10190
@[tracepoint:syscalls:sys_exit_openat, 0]: 10190
```

Experience: ps(1) failure

Which syscall is abnormally failing (without multi-probe)?

```
# bpftrace -e 't:raw_syscalls:sys_exit /comm == "ps"/ {
    @[args->id, args->ret > 0 ? 0 : - args->ret] = count(); }'
Attaching 1 probe...
[...]
@[21, 2]: 120
@[5, 0]: 180
@[10, 0]: 230
@[13, 0]: 240
@[9, 0]: 350
@[4, 0]: 5000
@[0, 0]: 10170
@[3, 0]: 10190
@[257, 0]: 10190
```

Experience: ps(1) failure

Which syscall is abnormally failing (without multi-probe)?

```
# bpftrace -e 't:raw_syscalls:sys_exit /comm == "ps"/ {
    @[ksym(*(kaddr("sys_call_table") + args->id * 8)),
      args->ret > 0 ? 0 : - args->ret] = count(); }'
[...]
@[sys_brk, 0]: 8202
@[sys_ioctl, 25]: 8203
@[sys_access, 2]: 32808
@[SyS_openat, 2]: 32808
@[sys_newfstat, 0]: 49213
@[sys_newstat, 2]: 60820
@[sys_mprotect, 0]: 62882
[...]
```

caught 1 extra failure

ioctl() was a dead end

Experience: ps(1) failure

Which syscall is *successfully* failing?

```
# bpftrace -e 't:syscalls:sys_exit_getdents /comm == "ps"/ {
    printf("ret: %d\n", args->ret); }'
[...]
ret: 9192
ret: 0
ret: 9216
ret: 0
ret: 9168
ret: 0
ret: 5640
ret: 0
^C
```

Experience: ps(1) failure

Which syscall is *successfully* failing?

```
# bpftrace -e 't:syscalls:sys_enter_getdents /comm == "ps"/ {
    @start[tid] = nsecs; }
  t:syscalls:sys_exit_getdents /@start[tid]/ {
    printf("%8d us, ret: %d\n", (nsecs - @start[tid]) / 1000,
    args->ret); delete(@start[tid]); }'
[...]
     559 us, ret: 9640
       3 us, ret: 0
     516 us, ret: 9576
       3 us, ret: 0
     373 us, ret: 7720
       2 us, ret: 0
^C
```

Experience: ps(1) failure

/proc debugging

```
# funccount '*proc*'
Tracing "*proc*"... Ctrl-C to end.^C
FUNC                              COUNT
[…]
proc_readdir                       1492
proc_readdir_de                    1492
proc_root_getattr                  1492
process_measurement                1669
kick_process                       1671
wake_up_process                    2188
proc_pid_readdir                   2984
proc_root_readdir                  2984
proc_fill_cache                  977263
```

Experience: ps(1) failure

Some quick dead ends

```
# bpftrace -e 'kr:proc_fill_cache /comm == "ps"/ {
    @[retval] = count(); }'
```

```
# bpftrace -e 'kr:nr_processes /comm == "ps"/ {
    printf("%d\n", retval); }'
```

```
# bpftrace -e 'kr:proc_readdir_de /comm == "ps"/ {
    printf("%d\n", retval); }'
```

```
# bpftrace -e 'kr:proc_root_readdir /comm == "ps"/ {
    printf("%d\n", retval); }'
```

Note: this is all in production

Experience: ps(1) failure

Getting closer to the cause

```
# bpftrace -e 'k:find_ge_pid /comm == "ps"/ { printf("%d\n", arg0);
}'
```

```
30707
31546
31913
31944
31945
31946
32070
```

success

failure

```
15020
15281
15323
15414
15746
15773
15778
```

Experience: ps(1) failure

find_ge_pid() entry argument & return value:

```
# bpftrace -e 'k:find_ge_pid /comm == "ps"/ { @nr[tid] = arg0; }
    kr:find_ge_pid /@nr[tid]/ {
      printf("%d: %llx\n", @nr[tid], retval); delete(@nr[tid]); }'
[…]
15561: ffff8a3ee70ad280
15564: ffff8a400244bb80
15569: ffff8a3f6f1a1840
15570: ffff8a3ffe890c00
15571: ffff8a3ffd23bdc0
15575: ffff8a40024fdd80
15576: 0
```

Experience: ps(1) failure

Kernel source:

```
struct pid *find_ge_pid(int nr, struct pid_namespace *ns)
{
        return idr_get_next(&ns->idr, &nr);
}
[…]
void *idr_get_next(struct idr *idr, int *nextid)
{
[…]
        slot = radix_tree_iter_find(&idr->idr_rt, &iter, id);
```

```
Subject [RFC 2/2] pid: Replace PID bitmap implementation with IDR API
Date    Sat, 9 Sep 2017 18:03:17 +0530
[…]
```

Experience: ps(1) failure

So far we have moved from:

ps(1) sometimes fails. Kernel bug!

To:

find_ge_pid() sometimes returns NULL instead of the next struct *pid

I'll keep digging after this keynote

Takeaway:

BPF enables better bug reports

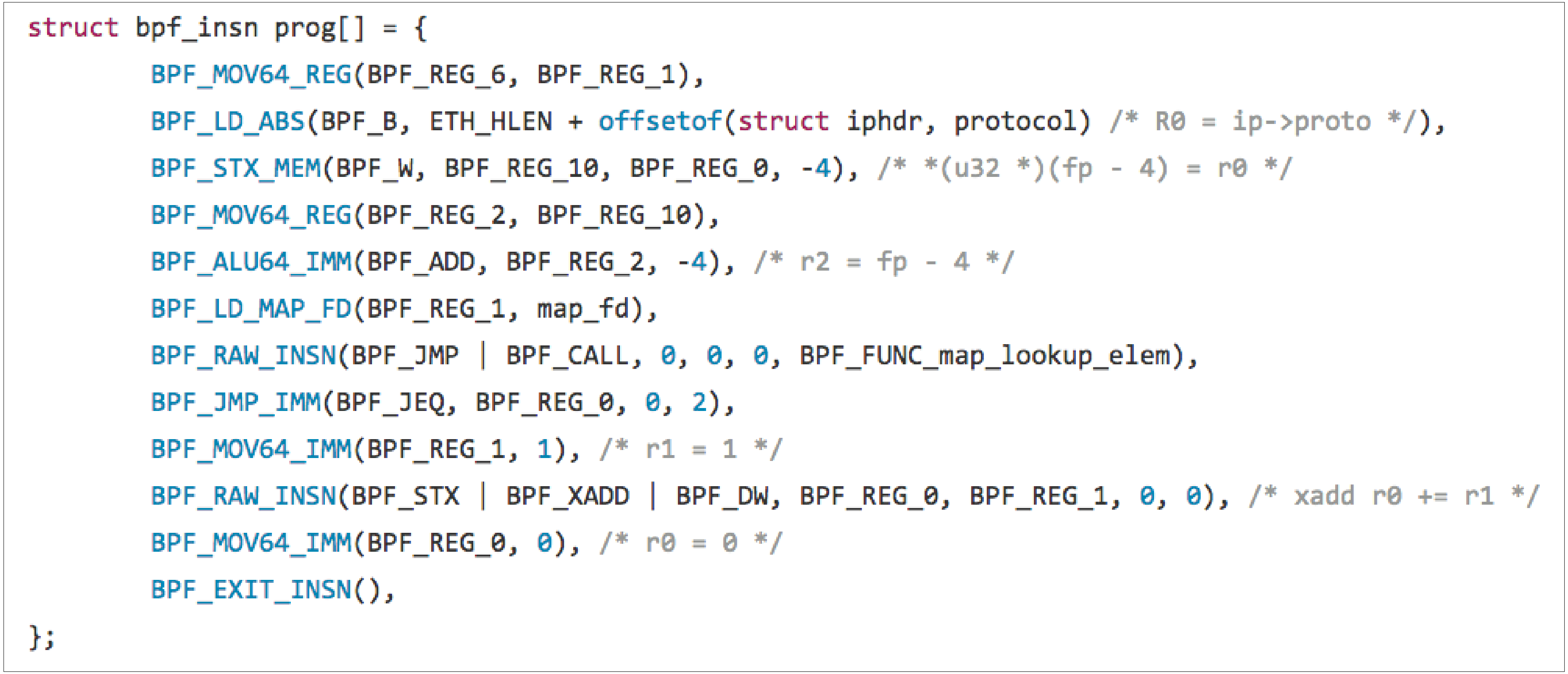
bpftrace: BPF observability front-end

```
# Files opened by process
bpftrace -e 't:syscalls:sys_enter_open { printf("%s %s\n", comm,
    str(args->filename)) }'

# Read size distribution by process
bpftrace -e 't:syscalls:sys_exit_read { @[comm] = hist(args->ret) }'

# Count VFS calls
bpftrace -e 'kprobe:vfs_* { @[func]++ }'

# Show vfs_read latency as a histogram
bpftrace -e 'k:vfs_read { @[tid] = nsecs }
    kr:vfs_read /@[tid]/ { @ns = hist(nsecs - @[tid]); delete(@tid) }’

# Trace user-level function
bpftrace -e 'uretprobe:bash:readline { printf(“%s\n”, str(retval)) }’
…
```

Linux 4.9+

https://github.com/iovisor/bpftrace

Raw BPF

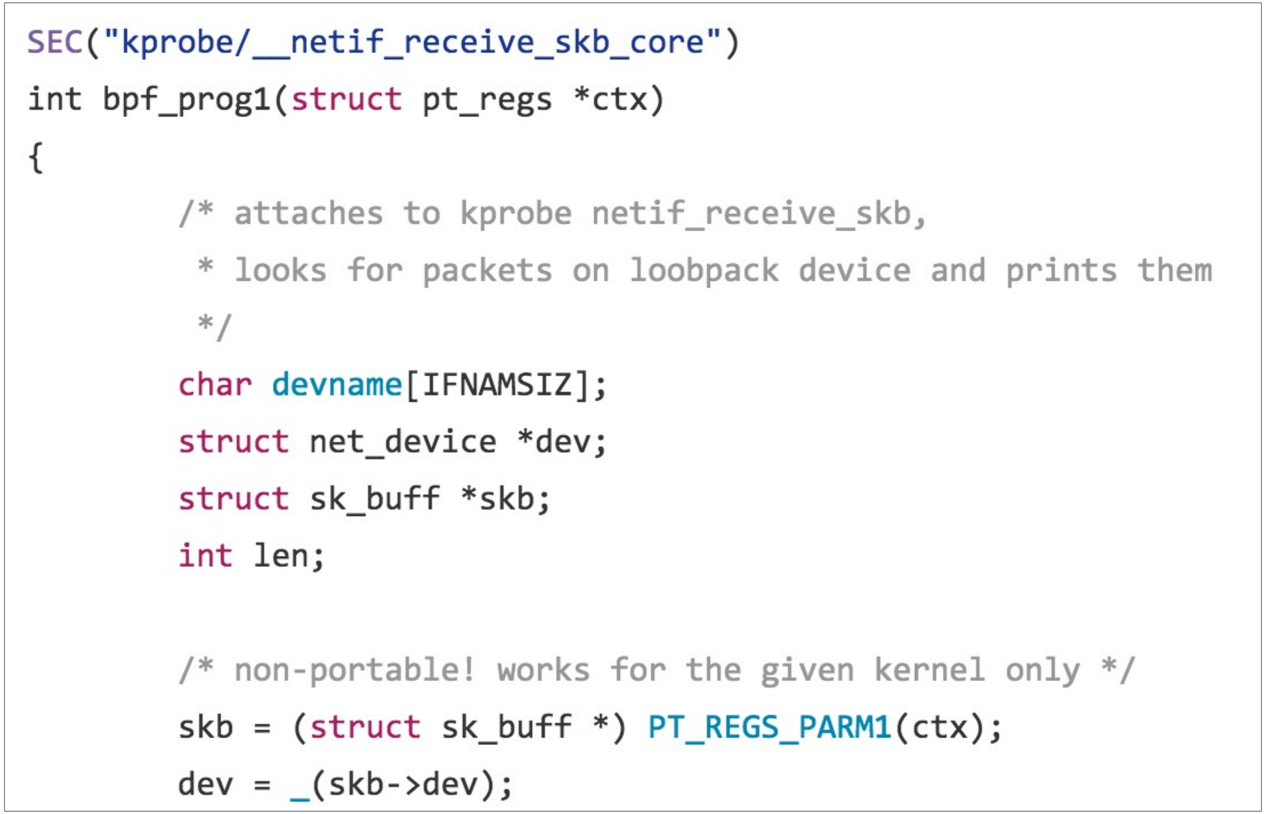


```
struct bpf_insn prog[] = {
        BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
        BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /* R0 = ip->proto */),
        BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
        BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
        BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
        BPF_LD_MAP_FD(BPF_REG_1, map_fd),
        BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
        BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
        BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
        BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
        BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
        BPF_EXIT_INSN(),
};
```

samples/bpf/sock_example.c

87 lines truncated

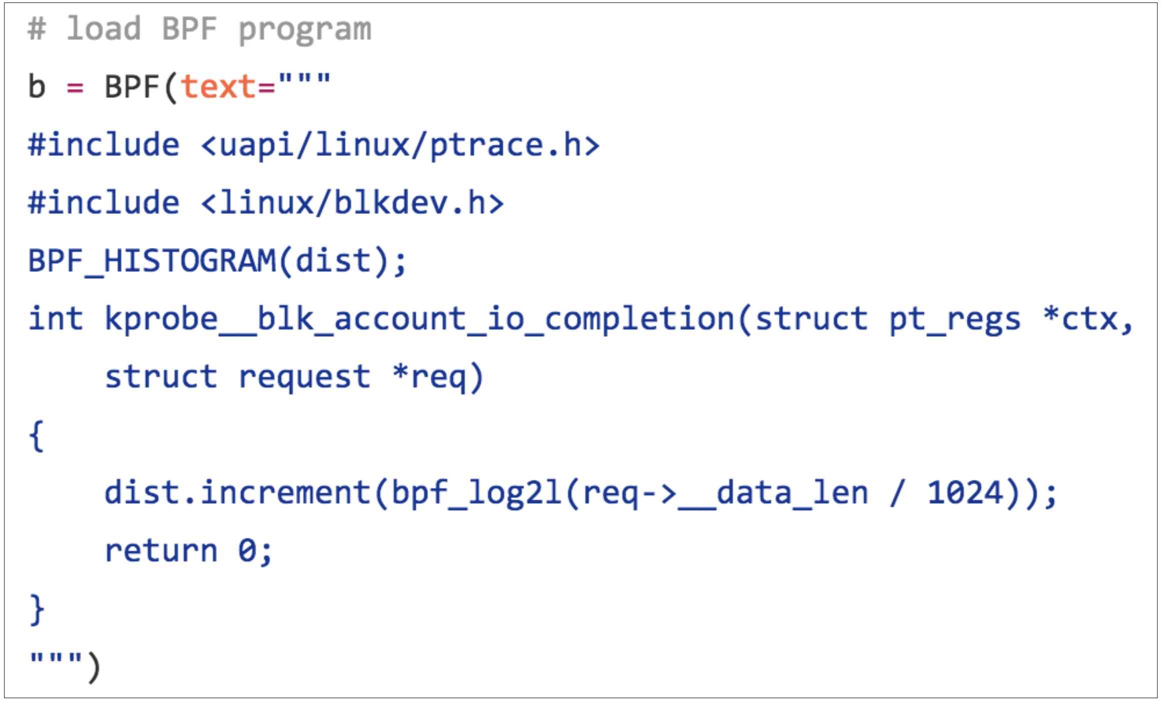
C/BPF



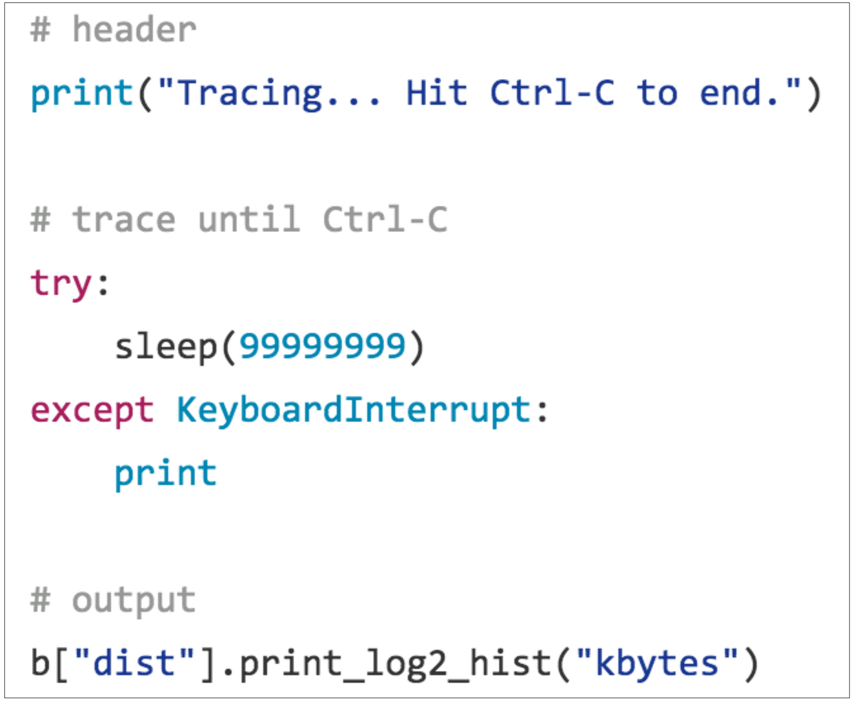
```
SEC("kprobe/__netif_receive_skb_core")
int bpf_prog1(struct pt_regs *ctx)
{
        /* attaches to kprobe netif_receive_skb,
         * looks for packets on loobpack device and prints them
         */
        char devname[IFNAMSIZ];
        struct net_device *dev;
        struct sk_buff *skb;
  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

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

samples/bpf/tracex1_kern.c
58 lines truncated

bcc/BPF (C & Python)



```
# load BPF program
b = BPF(text="""
#include <uapi/linux/ptrace.h>
#include <linux/blkdev.h>
BPF_HISTOGRAM(dist);
int kprobe__blk_account_io_completion(struct pt_regs *ctx,
    struct request *req)
{
    dist.increment(bpf_log2l(req->__data_len / 1024));
    return 0;
}
""")
```



```
# header
print("Tracing... Hit Ctrl-C to end.")

# trace until Ctrl-C
try:
    sleep(99999999)
except KeyboardInterrupt:
    print

# output
b["dist"].print_log2_hist("kbytes")
```

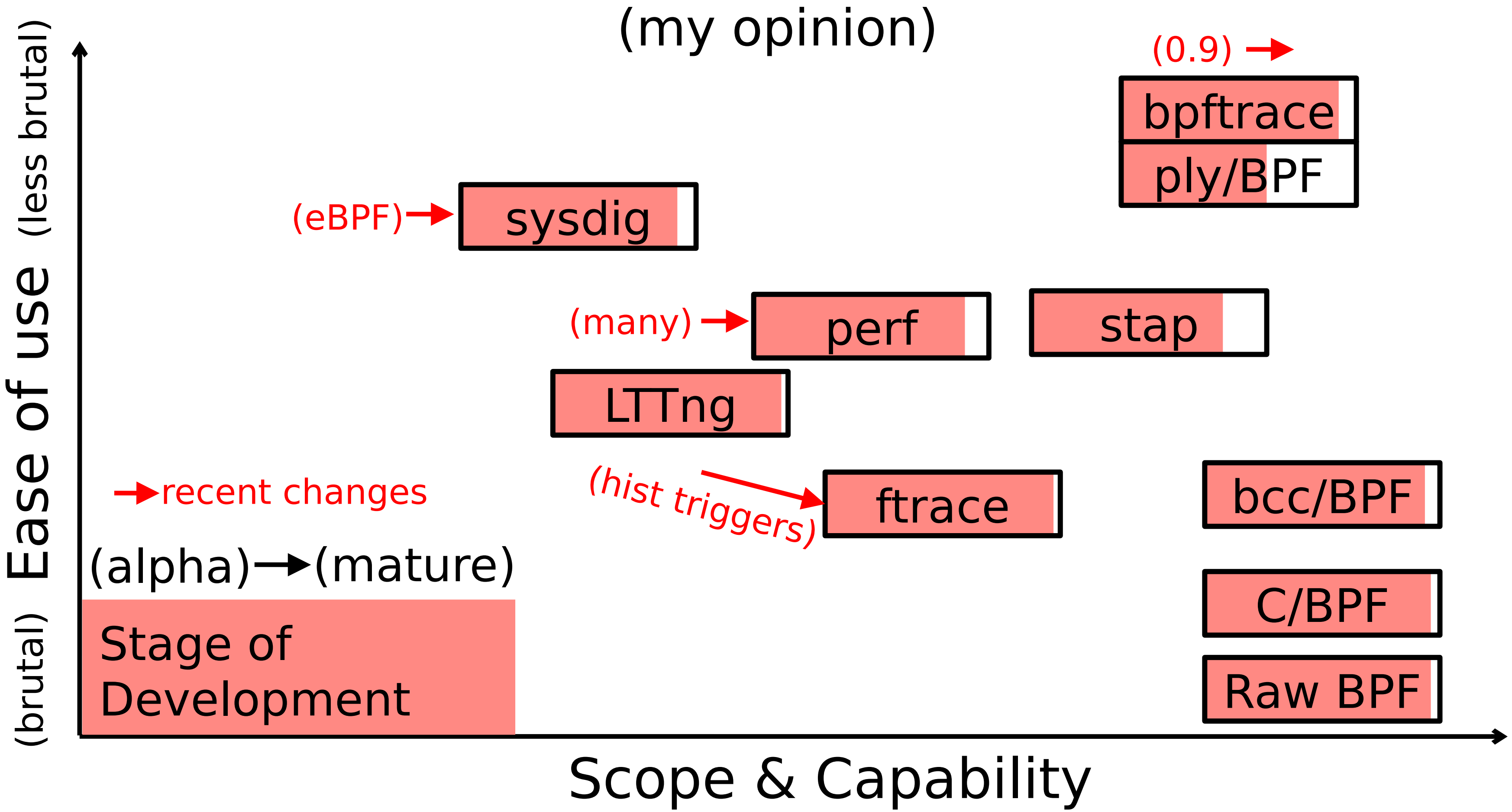
bcc examples/tracing/bitehist.py
entire program

bpftrace/BPF

```
bpftrace -e 'kr:vfs_read { @ = hist(retval); }'
```

https://github.com/iovisor/bpftrace
entire program

The Tracing Landscape, Apr 2019



Experience: readahead

Experience: readahead

Is readahead polluting the cache?

Experience: readahead

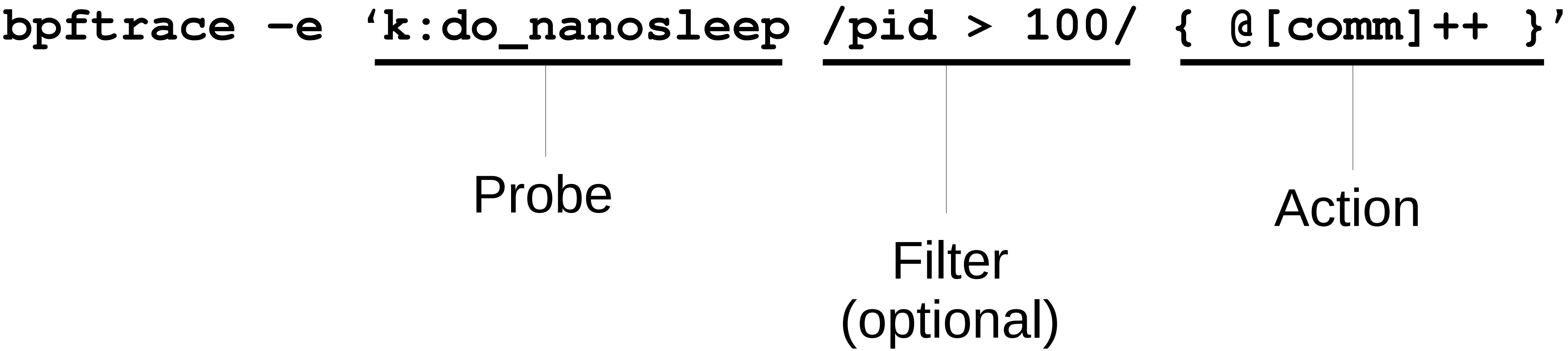
Is readahead polluting the cache?

```
# readahead.bt
Attaching 5 probes...
^C
Readahead unused pages: 128

Readahead used page age (ms):
@age_ms:
[1]                 2455 |@@@@@@@@@@@@@@@                                     |
[2, 4)              8424 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@|
[4, 8)              4417 |@@@@@@@@@@@@@@@@@@@@@@@@@@@                         |
[8, 16)             7680 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@     |
[16, 32)            4352 |@@@@@@@@@@@@@@@@@@@@@@@@@@                          |
[32, 64)               0 |                                                    |
[64, 128)              0 |                                                    |
[128, 256)           384 |@@                                                  |
#!/usr/local/bin/bpftrace

kprobe:__do_page_cache_readahead    { @in_readahead[tid] = 1; }
kretprobe:__do_page_cache_readahead { @in_readahead[tid] = 0; }

kretprobe:__page_cache_alloc
/@in_readahead[tid]/
{
    @birth[retval] = nsecs;
    @rapages++;
}

kprobe:mark_page_accessed
/@birth[arg0]/
{
    @age_ms = hist((nsecs - @birth[arg0]) / 1000000);
    delete(@birth[arg0]);
    @rapages--;
}

END
{
    printf("\nReadahead unused pages: %d\n", @rapages);
    printf("\nReadahead used page age (ms):\n");
    print(@age_ms); clear(@age_ms);
    clear(@birth); clear(@in_readahead); clear(@rapages);
}
```

Takeaway:

bpftrace is good for short tools

bpftrace Syntax



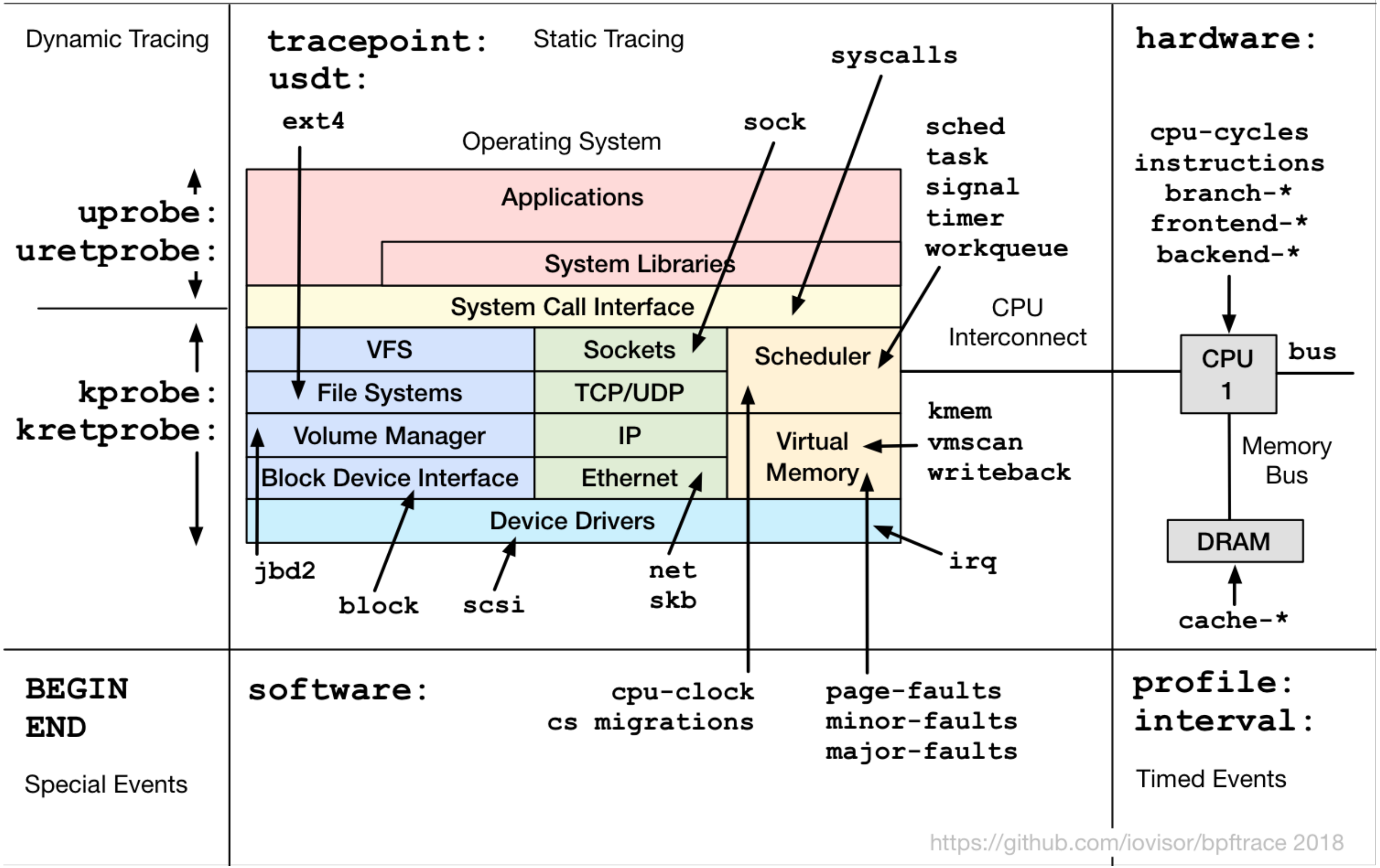
```
bpftrace -e ‘k:do_nanosleep /pid > 100/ { @[comm]++ }’
```

Probe

Filter
(optional)

Action

Probes



Dynamic Tracing
uprobe:
uretprobe:
kprobe:
kretprobe:
tracepoint:
usdt:
Static Tracing
syscalls
ext4
sock
Operating System
sched
task
signal
timer
workqueue
Applications
System Libraries
System Call Interface
VFS
Sockets
Scheduler
File Systems
TCP/UDP
Volume Manager
IP
Virtual Memory
Block Device Interface
Ethernet
Device Drivers
CPU Interconnect
kmem
vmscan
writeback
irq
jbd2
block
scsi
net
skb
hardware:
cpu-cycles
instructions
branch-*
frontend-*
backend-*
CPU 1
bus
Memory Bus
DRAM
cache-*
BEGIN
END
Special Events
software:
cpu-clock
cs migrations
page-faults
minor-faults
major-faults
profile:
interval:
Timed Events
https://github.com/iovisor/bpftrace 2018

Probe Type Shortcuts

`tracepoint`	`t`	Kernel static tracepoints
`usdt`	`U`	User-level statically defined tracing
`kprobe`	`k`	Kernel function tracing
`kretprobe`	`kr`	Kernel function returns
`uprobe`	`u`	User-level function tracing
`uretprobe`	`ur`	User-level function returns
`profile`	`p`	Timed sampling across all CPUs
`interval`	`i`	Interval output
`software`	`s`	Kernel software events
`hardware`	`h`	Processor hardware events

Filters

- `/pid == 181/`
- `/comm != “sshd”/`
- `/@ts[tid]/`

Actions

- Per-event output
  - **`printf()`**
  - **`system()`**
  - **`join()`**
  - **`time()`**
- Map Summaries
  - **`@ = count()`** or **`@++`**
  - **`@ = hist()`**
  - …

The following is in the https://github.com/iovisor/bpftrace/blob/master/docs/reference_

Functions

`hist(n)`	Log2 histogram
`lhist(n, min, max, step)`	Linear hist.
`count()`	Count events
`sum(n)`	Sum value
`min(n)`	Minimum value
`max(n)`	Maximum value
`avg(n)`	Average value
`stats(n)`	Statistics
`str(s)`	String
`ksym(p)`	Resolve kernel addr
`usym(p)`	Resolve user addr
`kaddr(n)`	Resolve kernel symbol
`uaddr(n)`	Resolve user symbol
`printf(fmt, ...)`	Print formatted
`print(@x[, top[, div]])`	Print map
`delete(@x)`	Delete map element
`clear(@x)`	Delete all keys/values
`reg(n)`	Register lookup
`join(a)`	Join string array
`time(fmt)`	Print formatted time
`system(fmt)`	Run shell command
`cat(file)`	Print file contents
`exit()`	Quit bpftrace

Variable Types

- Basic Variables
  - **`@global`**
  - **`@thread_local[tid]`**
  - **`$scratch`**
- Associative Arrays
  - **`@array[key] = value`**
- Buitins
  - **`pid`**
  - **`...`**

Builtin Variables

`pid`	Process ID (kernel tgid)
`tid`	Thread ID (kernel pid)
`cgroup`	Current Cgroup ID
`uid`	User ID
`gid`	Group ID
`nsecs`	Nanosecond timestamp
`cpu`	Processor ID
`comm`	Process name
`kstack`	Kernel stack trace
`ustack`	User stack trace

`arg0`, **`arg1`**, ...	Function args
`retval`	Return value
`args`	Tracepoint args
`func`	Function name
`probe`	Full probe name
`curtask`	Curr task_struct (u64)
`rand`	Random number (u32)

bpftrace: biolatency

```
#!/usr/local/bin/bpftrace

BEGIN
{
    printf("Tracing block device I/O... Hit Ctrl-C to end.\n");
}

kprobe:blk_account_io_start
{
    @start[arg0] = nsecs;
}

kprobe:blk_account_io_completion
/@start[arg0]/

{
    @usecs = hist((nsecs - @start[arg0]) / 1000);
    delete(@start[arg0]);
}
```

Experience: superping!

Experience: superping

How much is scheduler latency?

```
# ping 172.20.0.1
PING 172.20.0.1 (172.20.0.1) 56(84) bytes of data.
64 bytes from 172.20.0.1: icmp_seq=1 ttl=64 time=2.87 ms
64 bytes from 172.20.0.1: icmp_seq=2 ttl=64 time=1.66 ms
64 bytes from 172.20.0.1: icmp_seq=3 ttl=64 time=1.55 ms
64 bytes from 172.20.0.1: icmp_seq=4 ttl=64 time=1.11 ms
64 bytes from 172.20.0.1: icmp_seq=5 ttl=64 time=2.48 ms
64 bytes from 172.20.0.1: icmp_seq=6 ttl=64 time=2.39 ms
[...]
```

Experience: superping

How much is scheduler latency?

```
# ./superping.bt
Attaching 6 probes...
Tracing ICMP echo request latency. Hit Ctrl-C to end.
IPv4 ping, ID 9827 seq 1: 2883 us
IPv4 ping, ID 9827 seq 2: 1682 us
IPv4 ping, ID 9827 seq 3: 1568 us
IPv4 ping, ID 9827 seq 4: 1078 us
IPv4 ping, ID 9827 seq 5: 2486 us
IPv4 ping, ID 9827 seq 6: 2394 us
[...]
```

?!

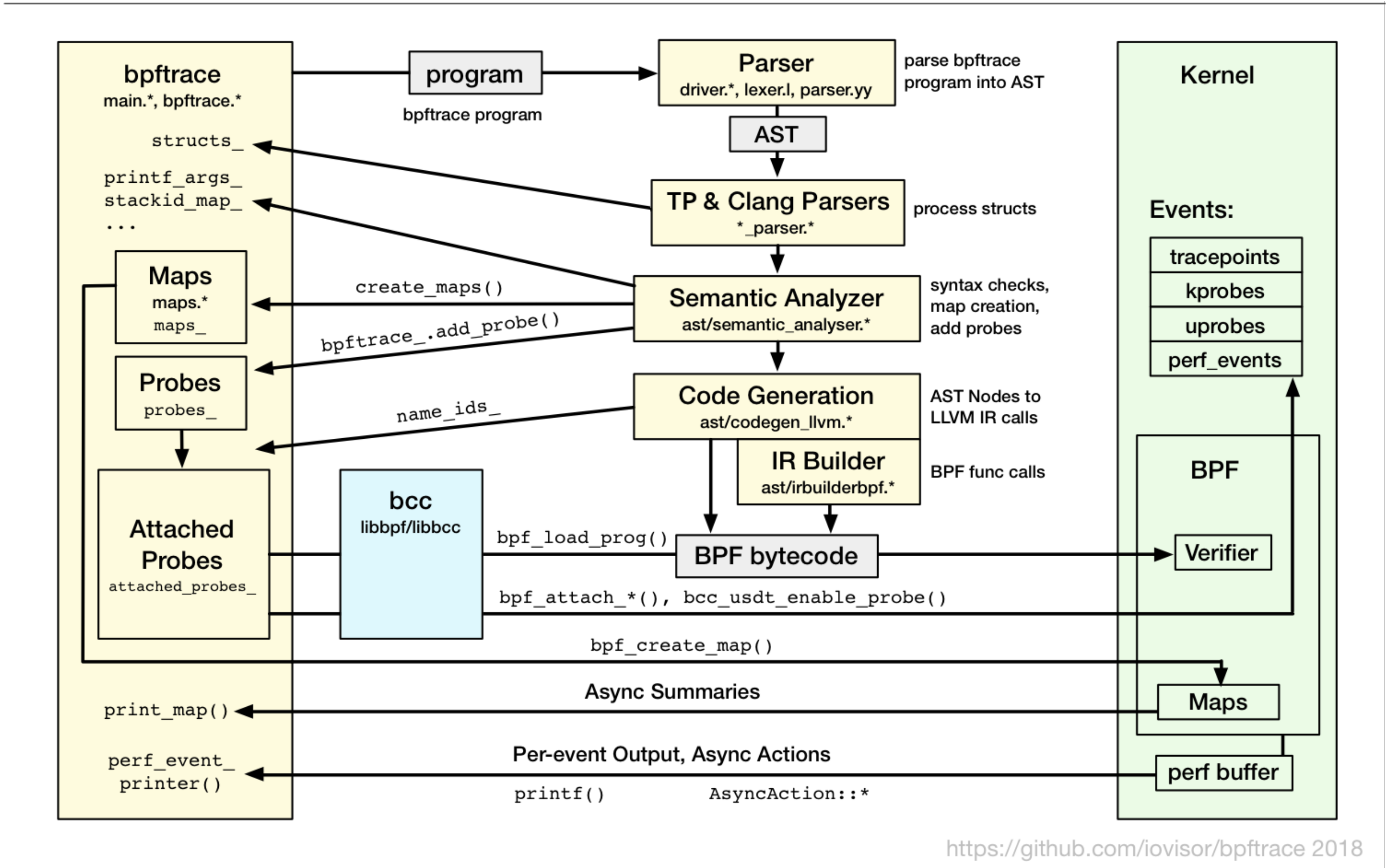
```
#!/usr/local/bin/bpftrace

#include <linux/skbuff.h>
#include <linux/icmp.h>
#include <linux/ip.h>
#include <linux/ipv6.h>
#include <linux/in.h>

BEGIN { printf("Tracing ICMP echo request latency. Hit Ctrl-C to end.\n"); }

kprobe:ip_send_skb
{
    $skb = (struct sk_buff *)arg1;
    // get IPv4 header; see skb_network_header():
    $iph = (struct iphdr *)($skb->head + $skb->network_header);
    if ($iph->protocol == IPPROTO_ICMP) {
        // get ICMP header; see skb_transport_header():
        $icmph = (struct icmphdr *)($skb->head + $skb->transport_header);
        if ($icmph->type == ICMP_ECHO) {
            $id = $icmph->un.echo.id;
            $seq = $icmph->un.echo.sequence;
            @start[$id, $seq] = nsecs;
        }
    }
}
[...]
```

Note: no debuginfo required

Takeaway:

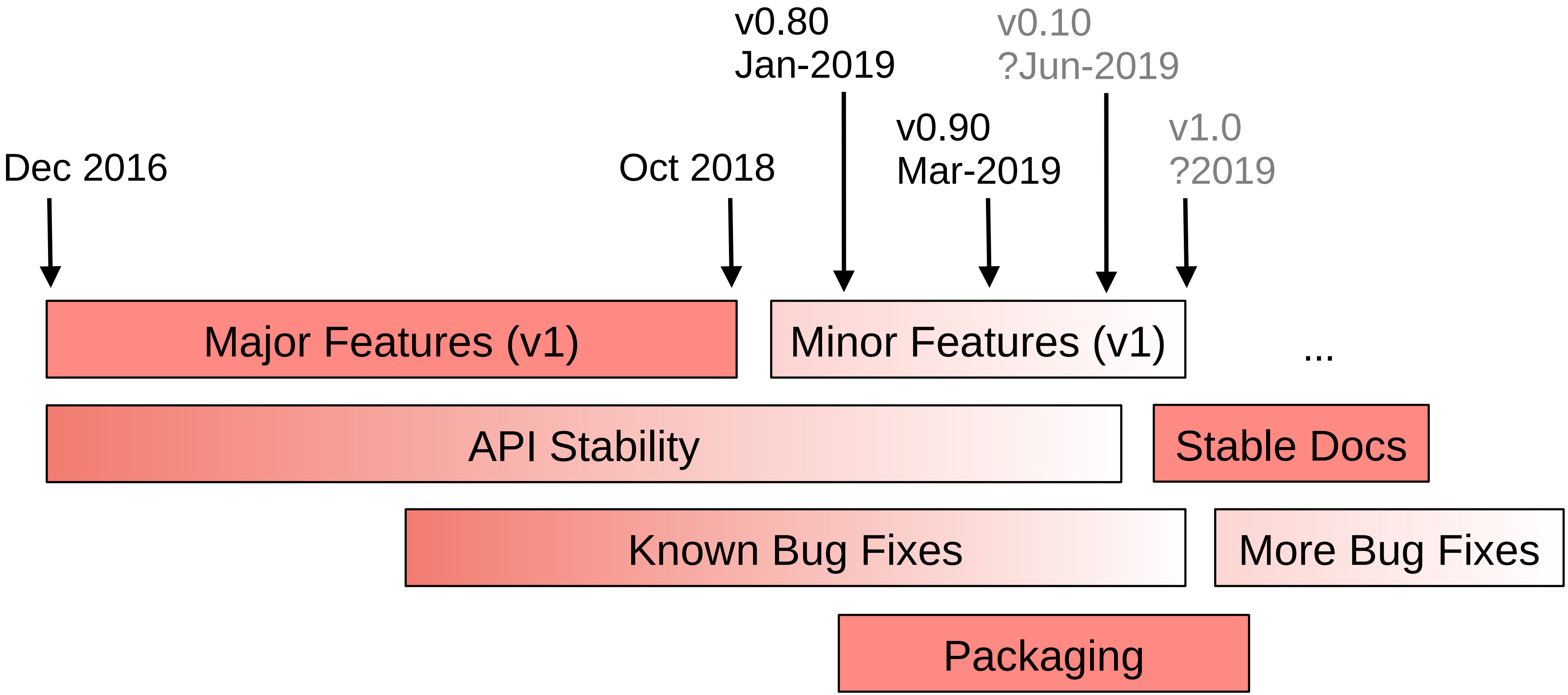
BPF tracing can walk structs

bpftrace Internals



https://github.com/iovisor/bpftrace 2018

bpftrace Development



Dec 2016

Oct 2018

v0.80
Jan-2019

v0.90
Mar-2019

v0.10
?Jun-2019

v1.0
?2019

Major Features (v1)

Minor Features (v1)

...

API Stability

Stable Docs

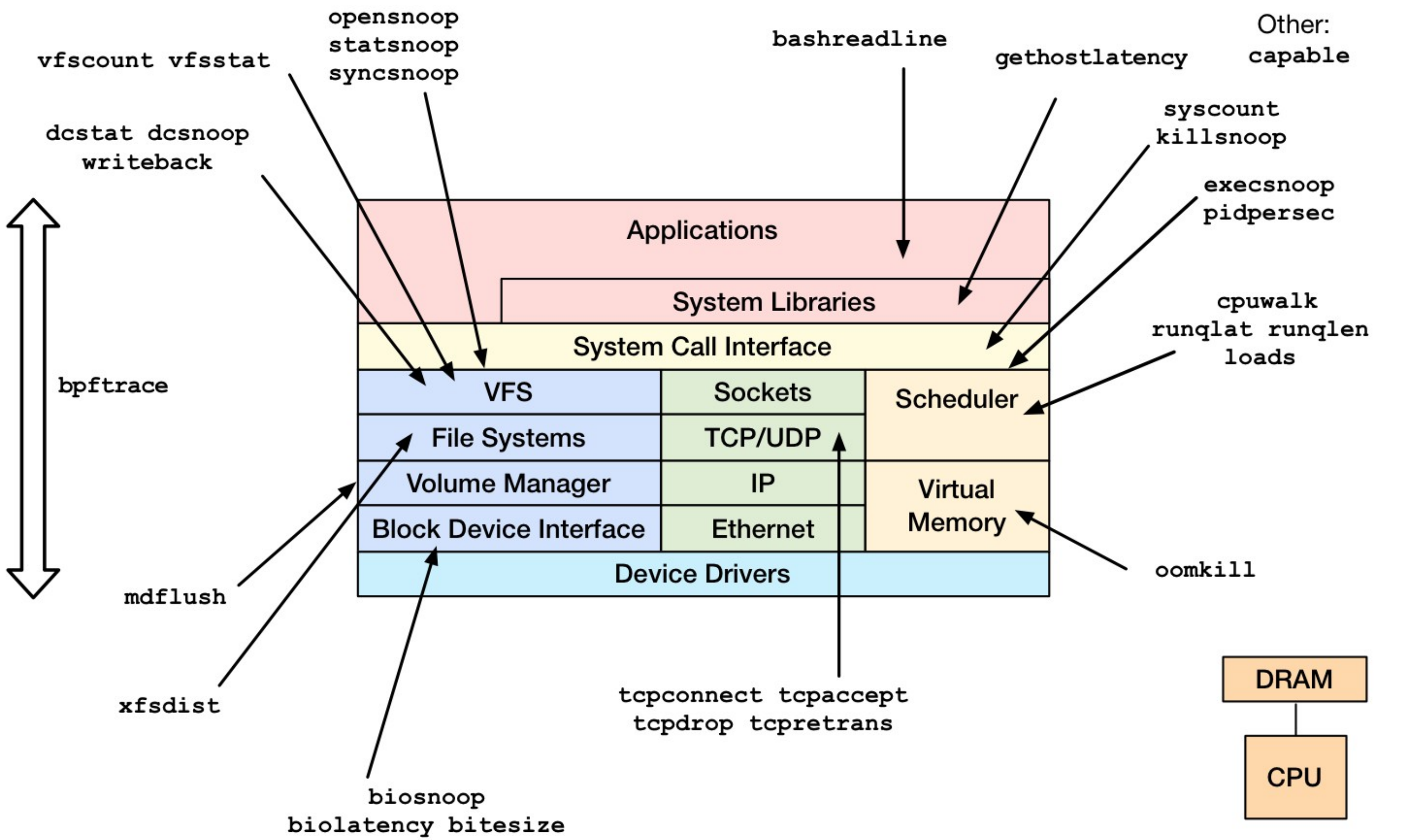
Known Bug Fixes

More Bug Fixes

Packaging

bpftrace Tools

Linux 4.9+



opensnoop
statsnoop
syncsnoop

vfscount vfsstat

bashreadline

gethostlatency

Other:
capable

dcstat dcsnoop
writeback

syscount
killsnoop

execsnoop
pidpersec

cpuwalk
runqlat runqlen
loads

bpftrace

Applications

System Libraries

System Call Interface

VFS

Sockets

Scheduler

File Systems

TCP/UDP

Volume Manager

IP

Virtual Memory

Block Device Interface

Ethernet

Device Drivers

oomkill

mdflush

xfsdist

tcpconnect tcpaccept
tcpdrop tcpretrans

DRAM

CPU

biosnoop
biolatency bitesize

https://github.com/

Experience: tcplife

Experience: tcplife

Which processes are connecting to which port?

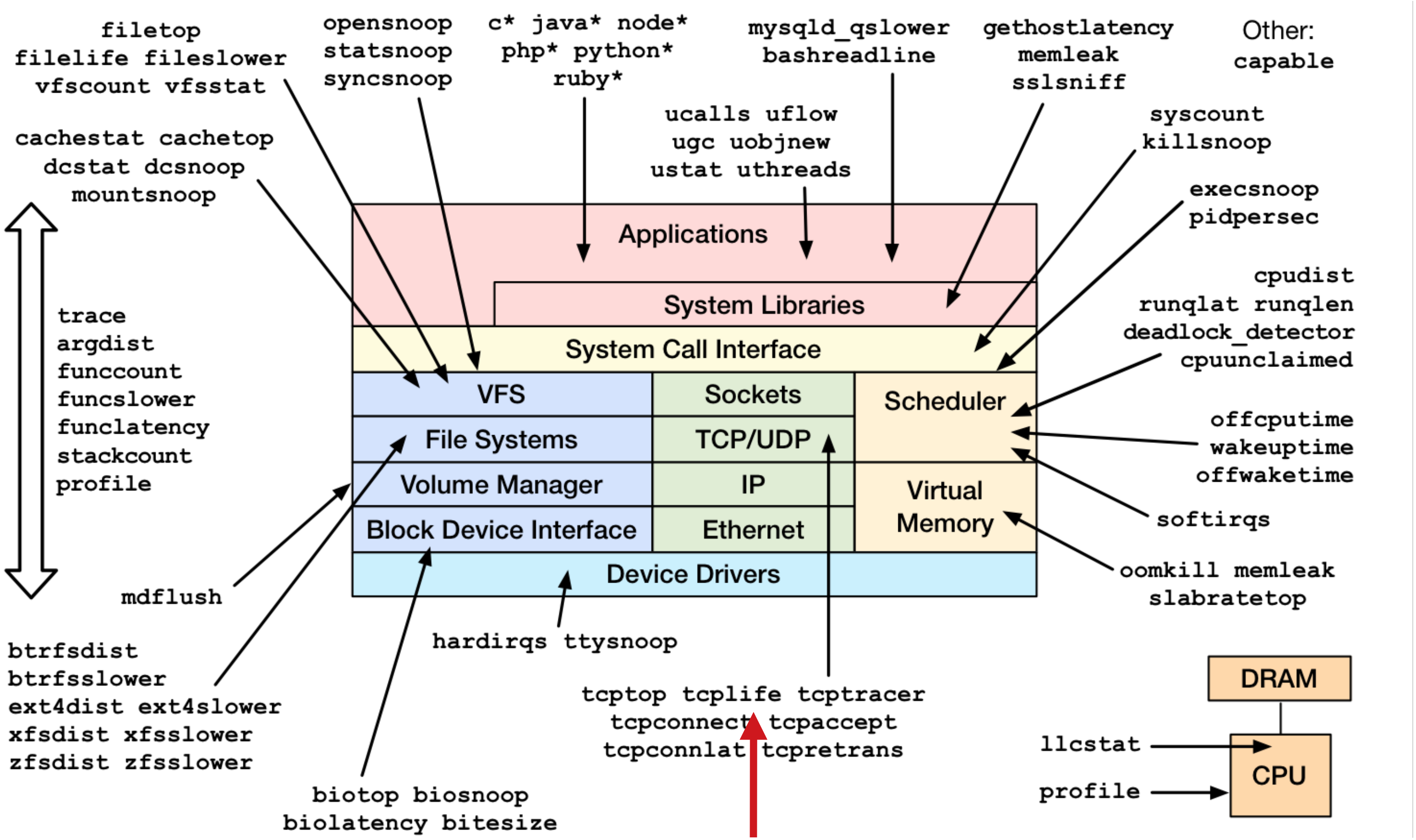
Experience: tcplife

Which processes are connecting to which port?

```
# ./tcplife
PID   COMM       LADDR           LPORT RADDR           RPORT TX_KB RX_KB MS
22597 recordProg 127.0.0.1       46644 127.0.0.1       28527     0     0 0.23
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46644     0     0 0.28
22598 curl       100.66.3.172    61620 52.205.89.26    80        0     1 91.79
22604 curl       100.66.3.172    44400 52.204.43.121   80        0     1 121.38
22624 recordProg 127.0.0.1       46648 127.0.0.1       28527     0     0 0.22
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46648     0     0 0.27
22647 recordProg 127.0.0.1       46650 127.0.0.1       28527     0     0 0.21
3277  redis-serv 127.0.0.1       28527 127.0.0.1       46650     0     0 0.26
[...]
```

bcc

Linux 4.4+



https://github.com/iovisor/bcc

bcc: tcplife

```
int kprobe__tcp_set_state(struct pt_regs *ctx, struct sock *sk, int state)
{
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    // lport is either used in a filter here, or later
    u16 lport = sk->__sk_common.skc_num;
[…]
    struct tcp_sock *tp = (struct tcp_sock *)sk;
    rx_b = tp->bytes_received;
    tx_b = tp->bytes_acked;

    u16 family = sk->__sk_common.skc_family;

    if (family == AF_INET) {
        struct ipv4_data_t data4 = {};
        data4.span_us = delta_us;
        data4.rx_b = rx_b;
        data4.tx_b = tx_b;
        data4.ts_us = bpf_ktime_get_ns() / 1000;
        data4.saddr = sk->__sk_common.skc_rcv_saddr;
        data4.daddr = sk->__sk_common.skc_daddr;
[…]
```

Experience: tcplife

From kprobes to tracepoints

```
# bpftrace -lv t:tcp:tcp_set_state
tracepoint:tcp:tcp_set_state
    const void * skaddr;
    int oldstate;
    int newstate;
    __u16 sport;
    __u16 dport;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
```

```
# bpftrace -lv t:sock:inet_sock_set_state
tracepoint:sock:inet_sock_set_state
    const void * skaddr;
    int oldstate;
    int newstate;
    __u16 sport;
    __u16 dport;
    __u16 family;
    __u8 protocol;
    __u8 saddr[4];
    __u8 daddr[4];
    __u8 saddr_v6[16];
    __u8 daddr_v6[16];
```

Linux 4.15 → **Linux 4.16+**

Takeaways:

bcc for complex tools

kprobes can prototype tracepoints

tracepoints can change (best effort)

Experience: cachestat

Experience: cachestat

What is the page cache hit ratio?

Experience: cachestat

What is the page cache hit ratio?

```
# cachestat
    HITS   MISSES  DIRTIES HITRATIO   BUFFERS_MB  CACHED_MB
    1132        0        4  100.00%          277       4367
     161        0       36  100.00%          277       4372
      16        0       28  100.00%          277       4372
   17154    13750       15   55.51%          277       4422
      19        0        1  100.00%          277       4422
      83        0       83  100.00%          277       4421
      16        0        1  100.00%          277       4423
[...]
```

```
b.attach_kprobe(event="add_to_page_cache_lru", fn_name="do_count")
b.attach_kprobe(event="mark_page_accessed", fn_name="do_count")
b.attach_kprobe(event="account_page_dirtied", fn_name="do_count")
b.attach_kprobe(event="mark_buffer_dirty", fn_name="do_count")
[…]
    # total = total cache accesses without counting dirties
    # misses = total of add to lru because of read misses
    total = mpa - mbd
    misses = apcl - apd
    if misses < 0:
        misses = 0
    if total < 0:
        total = 0
    hits = total - misses

    # If hits are < 0, then its possible misses are overestimated
    # due to possibly page cache read ahead adding more pages than
    # needed. In this case just assume misses as total and reset hits.
    if hits < 0:
        misses = total
        hits = 0
[…]
```

This is a sandcastle

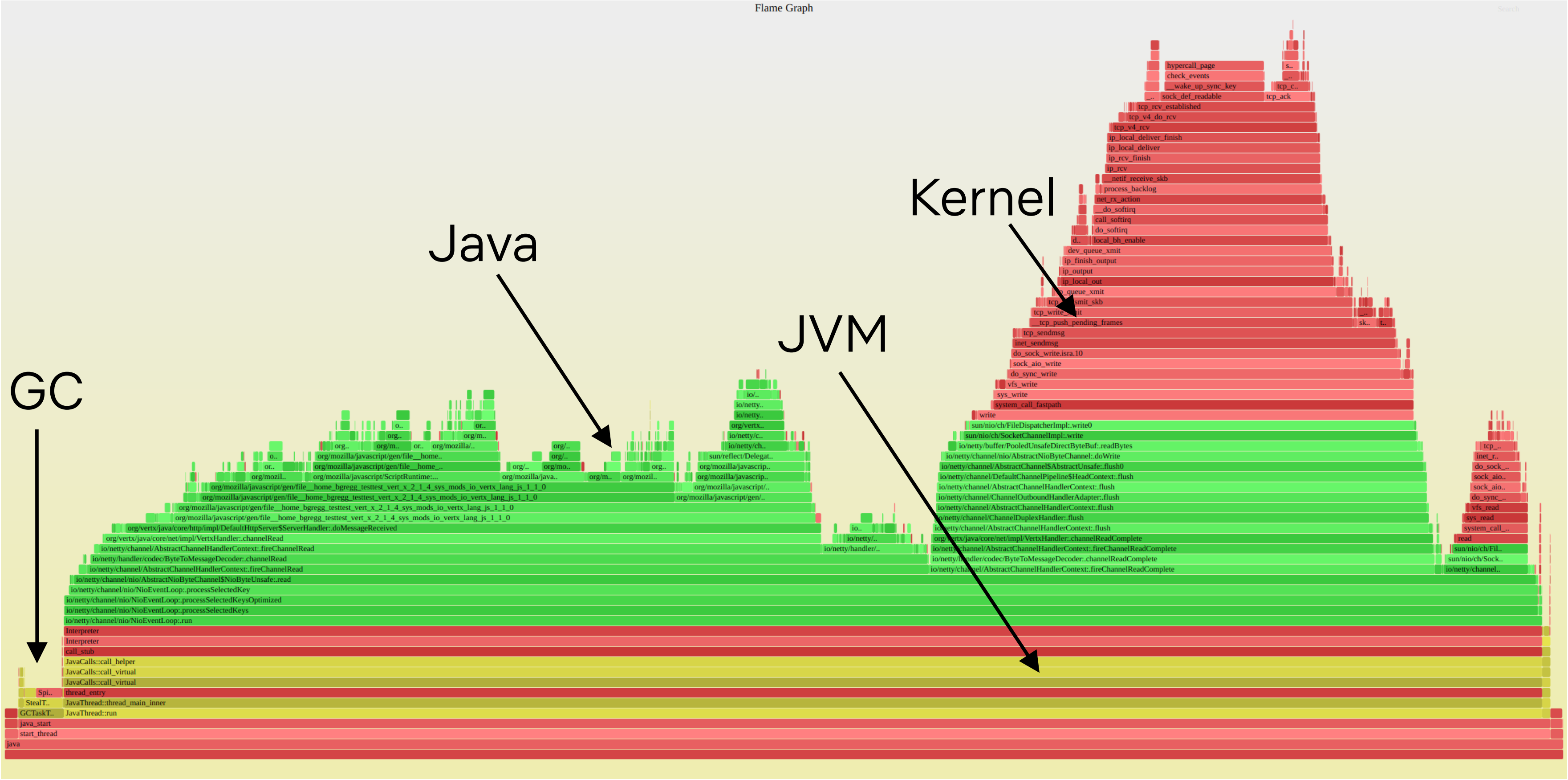
Takeaway:

BPF tracing can prototype /proc stats

Reality Check

Many of our perf wins are from CPU flame graphs not CLI tracing

CPU Flame Graphs



Kernel

Java

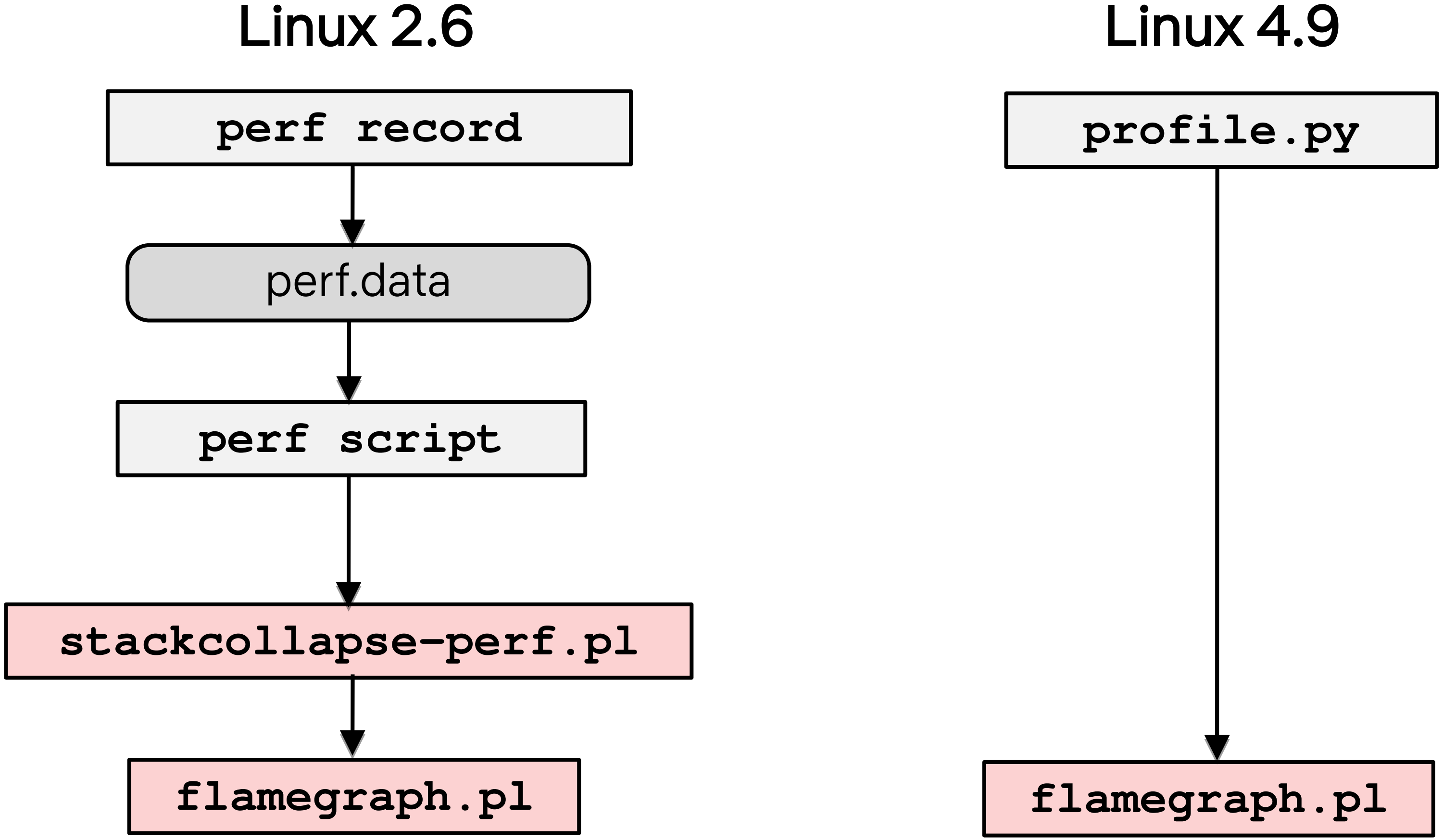
JVM

GC

Stack depth (0 - max)

Alphabetical frame sort (A - Z)

BPF-based CPU Flame Graphs



Takeaway:

BPF all the things!

Take Aways

- BPF observability:
  - bpftrace: one-liners, short scripts
  - bcc: complex tools
  - Production safe, and no debuginfo needed
- kprobe tools can prototype tracepoints, /proc stats
- I’m ok with tracepoints are best effort
- BPF all the things!

URLs

- https://github.com/iovisor/bpftrace
  - https://github.com/iovisor/bpftrace/blob/master/docs/tutorial_one_liners.md
  - https://github.com/iovisor/bpftrace/blob/master/docs/reference_guide.md
- https://github.com/iovisor/bcc
  - https://github.com/iovisor/bcc/blob/master/docs/tutorial.md
  - https://github.com/iovisor/bcc/blob/master/docs/reference_guide.md
- http://www.brendangregg.com/ebpf.html

Update: this keynote was summarized by

https://lwn.net/Articles/787131/

Thanks

- bpftrace
  - Alastair Robertson (creator)
  - Netflix: myself, Mary Marchini
  - Sthima: Willian Gaspar
  - Facebook: Jon Haslam, Dan Xu
  - Augusto Mecking Caringi, Dale Hamel, ...
- eBPF & bcc
  - Facebook: Alexei Starovoitov, Teng Qin, Yonghong Song, Martin Lau, Mark Drayton, …
  - Netflix: myself
  - VMware: Brenden Blanco
  - Daniel Borkmann, David S. Miller, Sasha Goldsthein, Paul Chaignon, ...