BPF at Facebook

Alexei Starovoitov

Agenda

1) kernel upgrades
2) BPF in the datacenter
3) BPF evolution
4) where you can help

Kernel upgrades in FB

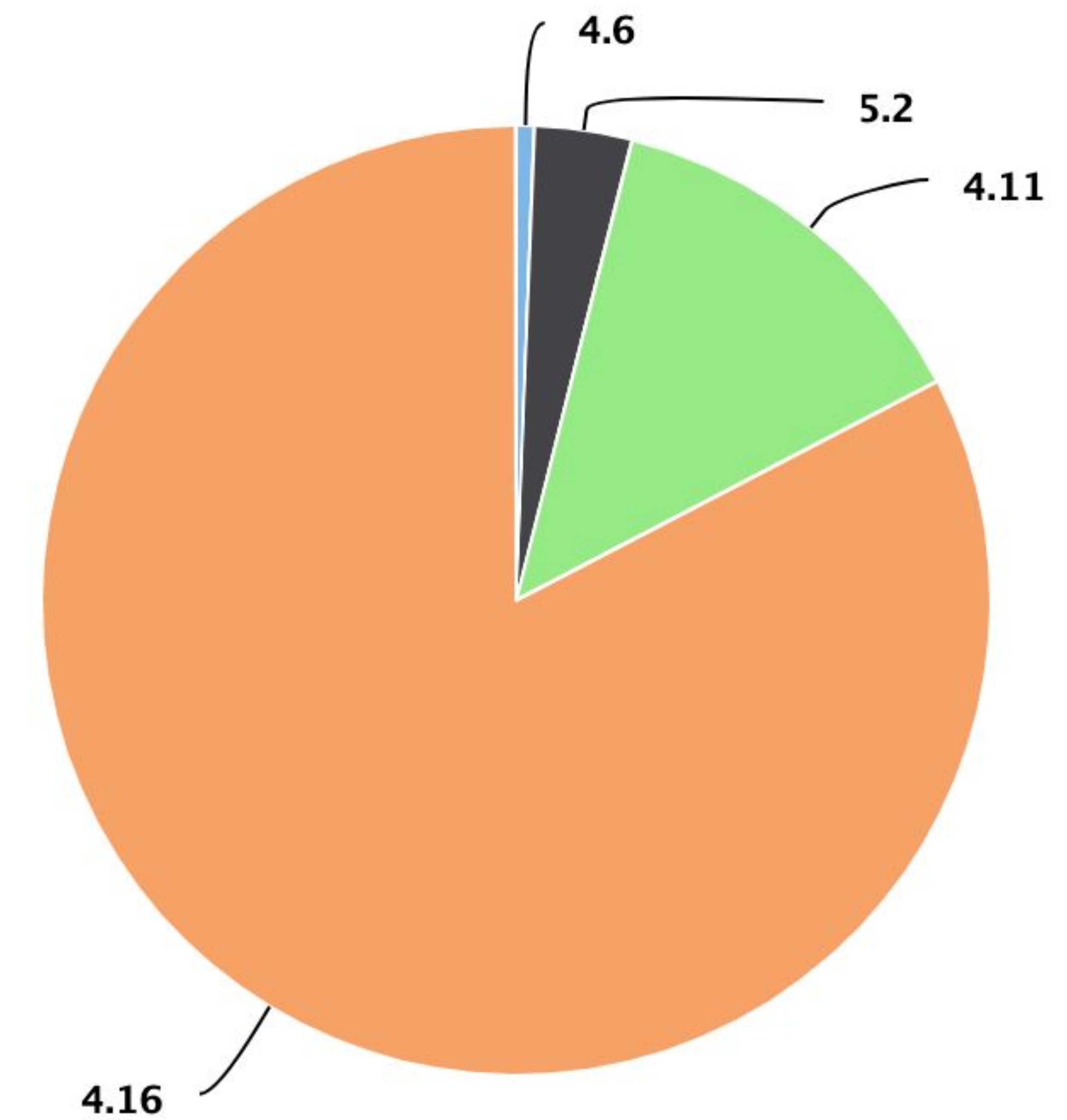
move fast

- "Upstream first" philosophy.
- Close to zero private patches.
- As soon as practical kernel team:
  . takes the latest upstream kernel
  . stabilizes it
  . rolls it across the fleet
  . backports relevant features until the cycle repeats
- It used to take months to upgrade. Now few weeks. Days when necessary.

Kernel version by count

- As of September 2019.
- It will be different tomorrow.

- One kernel version on most servers.
- Many 4.16.x flavors due to long tail.
- Challenging environment for user space.
- Even more challenging for BPF based tracing.



Do not break user space

"The first rule" of kernel programming... multiplied by FB scale.

- Must not change kernel ABI.
- Must not cause performance regressions.
- Must not change user space behavior.

- Investigate all differences.
    . Either unexpected improvement or regression.
    . Team work is necessary to root cause.

Do you use BPF?

- Run this command on your laptop:

```
sudo bpftool prog show | grep name | wc -l
```

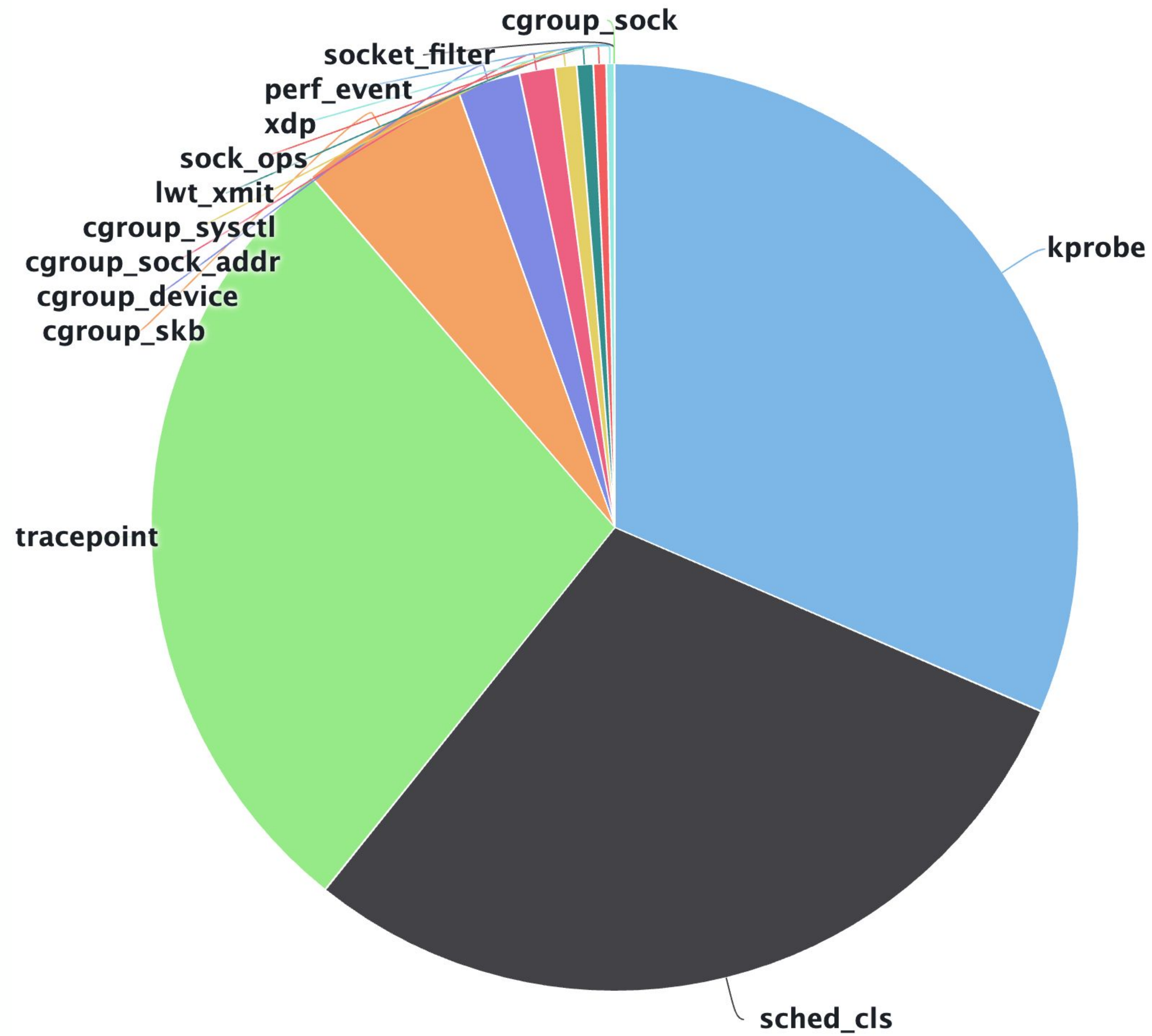
- What number does it print?
- Don't have bpftool ? Run this:

```
ls -la /proc/*/fd | grep bpf-prog | wc -l
```

BPF at Facebook

- ~40 BPF programs active on every server.
- ~100 BPF programs loaded on demand for short period of time.
- Mainly used by daemons that run on every server.
- Many teams are writing and deploying them.

BPF program distribution by type



Kernel team is involved in lots of investigations.

It's not true, but I often feel this way :)

Example 1: packet capture daemon

- This daemon is using SCHED_CLS BPF program.
- The program is attached to TC ingress and runs on every packet.
- With 1 out of million probability it does bpf_perf_event_output(skb).
- On new kernel this daemon causes 1% cpu regression.

- Disabling the daemon makes the regression go away.
- Is it BPF?

Example 1: packet capture daemon (resolved)

- Turned out the daemon is loading KPROBE BPF program as well for unrelated logic.
- kprobe-d function doesn't exist in new kernel.
- Daemon decides that BPF is unusable and falls back to NFLOG-based packet capture.
- nflog loads iptable modules and causes 1% cpu regression.

Takeaway for developers

- kprobe is not a stable ABI.
- Everytime kernel developers change the code some kernel developers pay the price.

Example 2: performance profiling daemon

- The daemon is using BPF tracepoints, kprobes in the scheduler and task execution.
- It collects kernel and user stack traces, walks python user stacks inside BPF program and aggregates across the fleet.
- This daemon is #1 tool for performance analysis.
- On new kernel it causes 2% cpu regression.
- Higher softirq times. Slower user apps.

- Disabling the daemon makes the regression go away.
- Is it BPF?

Example 2: performance profiling daemon (resolved)

- Turned out that simply installing kprobe makes 5.2 kernel remap kernel .text from 2M huge pages into 4k.
- That caused more I-TLB misses.
- Making BPF execution in the kernel slower and user space as well.

Takeaway

- kprobe is essential part of kernel functionality.

Example 3: security monitoring daemon

- The daemon is using 3 kprobes and 1 kretprobe.
- Its BPF program code just over 200 lines of C.
- It runs with low priority.
- It wakes up every few seconds, consumes 0.01% of one cpu and 0.01% of memory.
- Yet it causes large P99 latency regression for database server that runs on all other cpus and consumes many Gbytes of memory.
- Throughput of the database is not affected.

- Disabling the daemon makes the regression go away.
- Is it BPF?

Investigation

Facts:
- Occasionally memcpy() in a database gets stuck for 1/4 of a second.
- The daemon is rarely reading /proc/pid/environ.

Guesses:
- Is database waiting on kernel to handle page fault ?
- While kernel is blocked on mmap_sem ?
- but "top" and others read /proc way more often. Why that daemon is special?

- Dive into kernel code

```
fs/proc/base.c
    environ_read()
        access_remote_vm()
            down_read(&mm->mmap_sem)
```

funclatency.py - Time functions and print latency as a histogram

```
# funclatency.py -d100 -m __access_remote_vm
Tracing 1 functions for "__access_remote_vm"... Hit Ctrl-C to end.

     msecs               : count     distribution
         0 -> 1          : 21938    |****************************************|
         2 -> 3          : 0        |                                        |
         4 -> 7          : 0        |                                        |
         8 -> 15         : 0        |                                        |
        16 -> 31         : 0        |                                        |
        32 -> 63         : 0        |                                        |
        64 -> 127        : 0        |                                        |
       128 -> 255        : 7        |                                        |
       256 -> 511        : 3        |                                        |
Detaching...
```

This histogram shows that over the last 100 seconds there were 3 events where reading /proc took more than 256 ms.

funcslower.py - Dump kernel and user stack when given kernel function was slower than threshold

```
# funcslower.py -m 200 -KU __access_remote_vm
Tracing function calls slower than 200 ms... Ctrl+C to quit.
COMM            PID      LAT(ms)             RVAL FUNC
security_daemon 1720415  399.02              605 __access_remote_vm
    kretprobe_trampoline
    read
    facebook::...::readBytes(folly::File const&)
    ...
```

This was the kernel+user stack trace when our security daemon was stuck in sys_read() for 399 ms.

Yes. It's that daemon causing database latency spikes.

Collect more stack traces with offwaketime.py ...

```
finish_task_switch
__schedule
preempt_schedule_common
_cond_resched
__get_user_pages
get_user_pages_remote
__access_remote_vm
proc_pid_cmdline_read
__vfs_read
vfs_read
sys_read
do_syscall_64
read
facebook::...::readBytes(folly::File const&)
```

The task reading from /proc/pid/cmdline can go to sleep without releasing mmap_sem of mm of that pid.

The page fault in that pid will be blocked until this task finishes reading /proc.

Root cause

- The daemon is using 3 kprobes and 1 kretprobe.
- Its BPF program code just over 200 lines of C.
- It runs with low priority.
- It wakes up every few seconds, consumes 0.01% of one cpu and 0.01% of memory.

Low CPU quota for the daemon coupled with aggressive sysctl kernel.sched_* tweaks were responsible.

Takeaway

- BPF tracing tools are the best to tackle BPF regression.

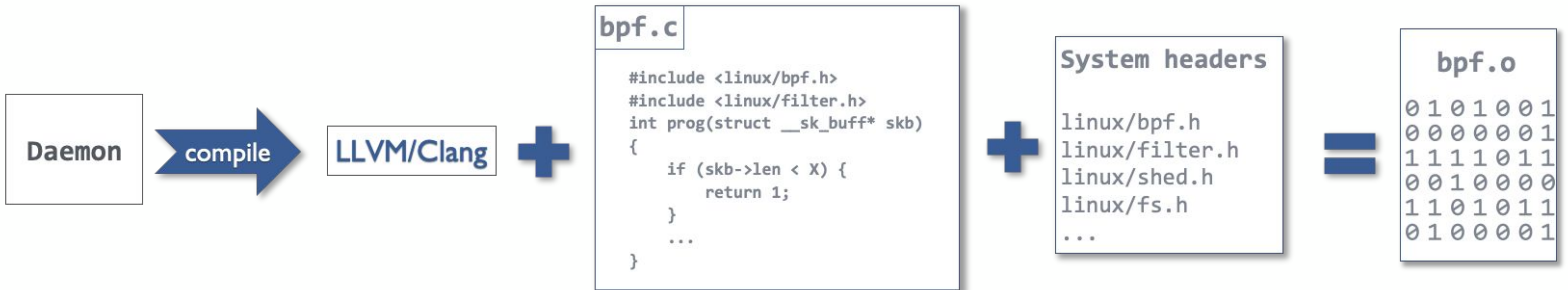
Another kind of BPF investigations

- Many kernels run in the datacenter.
- Daemons (and their BPF programs) need to work on all of them.

- BPF program works on developer server, but fails in production.

On developer server

On production server



- Embedded LLVM is safer than standalone LLVM.
- LLVM takes 70 Mb on disk. 20 Mb of memory at steady state. More at peak.
- Dependency on system kernel headers. Subsystem internal headers are missing.
- Compilation errors captured at runtime.
- Compilation on production server disturbs the main workload.
- And the other way around. llvm may take minutes to compile 100 lines of C.

BPF CO-RE (Compile Once Run Everywhere)

- Compile BPF program into "Run Everywhere" .o file (BPF assembly + extra).
- Test it on developer server against many "kernels".
- Adjust .o file on production server by libbpf.
- No compilation on production server.

BTF (BPF Type Format)

- BTF describes types, relocations, source code.

- LLVM compiles BPF program C code into BPF assembler and BTF.
- gcc+pahole compiles kernel C code into vmlinux binary and BTF.

- libbpf compares prog's BTF with vmlinux's BTF and adjusts BPF assembly before loading into the kernel.

- Developers can compile and test for kprobe and kernel data structure compatibility on a single server at build time instead of on N servers at run-time.

trace_kfree_skb today

```
#include <linux/skbuff.h>        /* kernel headers */
#include <linux/netdevice.h>
#include <uapi/linux/bpf.h>
#include <linux/version.h>
#include "bpf_helpers.h"

#define _(P) ({ typeof(P) val = 0; \
                bpf_probe_read(&val, sizeof(val), &P); \
                val; })

SEC("kprobe/kfree_skb")
int trace_kfree_skb(struct pt_regs *ctx)
{
    struct sk_buff * skb = (struct sk_buff *) PT_REGS_PARM1(ctx);
    struct net_device *dev = _(skb->dev);
    int ifindex = _(dev->ifindex);

    bpf_printk("skb->len %d\n", _(skb->len));
    bpf_printk("skb->queue_mapping %d\n", _(skb->queue_mapping));
    bpf_printk("dev->ifindex %d\n", ifindex);

    return 0;
}
```

clang -I/path_to_kernel_headers/ -I/path_to_user/

PARM2 typo will "work" too

Any type cast is allowed

six bpf_probe_read() calls

trace_kfree_skb today

```c
#include <linux/skbuff.h>       /* kernel headers */
#include <linux/netdevice.h>
#include <uapi/linux/bpf.h>
#include <linux/version.h>
#include "bpf_helpers.h"

#define _(P) ({ typeof(P) val = 0; \
                bpf_probe_read(&val, sizeof(val), &P); \
                val; })

SEC("kprobe/kfree_skb")
int trace_kfree_skb(struct pt_regs *ctx)
{
    struct sk_buff * skb = (struct sk_buff *) PT_REGS_PARM1(ctx);
    struct net_device *dev = _(skb->dev);
    int ifindex = _(dev->ifindex);

    bpf_printk("skb->len %d\n", _(skb->len));
    bpf_printk("skb->queue_mapping %d\n", _(skb->queue_mapping));
    bpf_printk("dev->ifindex %d\n", ifindex);

    return 0;
}
```

trace_kfree_skb with CO-RE

```c
#include <linux/bpf.h>
/* bpftool btf dump file \
 * /sys/kernel/btf/vmlinux format c > vmlinux.h */
#include "vmlinux.h"
#include "bpf_helpers.h"

#define _(P) (*__builtin_preserve_access_index(&P))

struct trace_kfree_skb {
    struct sk_buff *skb;
    void *location;
};

SEC("raw_tracepoint/trace_kfree_skb")
int trace_kfree_skb(struct trace_kfree_skb* ctx)
{
    struct sk_buff *skb = ctx->skb;
    struct net_device *dev = _(skb->dev);
    int ifindex = _(dev->ifindex);

    bpf_printk("skb->len %d\n", _(skb->len));
    bpf_printk("skb->queue_mapping %d\n", _(skb->queue_mapping));
    bpf_printk("dev->ifindex %d\n", ifindex);

    return 0;
}
```

Define kernel structs by hand instead of including vmlinux.h

If skb and location are accidentally swapped the verifier will catch it

Works with any raw tracepoint

Same kernel helper as in networking programs

```
#include <linux/bpf.h>
#include "bpf_helpers.h"
#define _(P) (*__builtin_preserve_access_index(&P))

struct net_device {                         /* same as kernel's struct net_device */
    int ifindex;
};
struct sk_buff { /* field names and sizes should match to those in the kernel */
    unsigned int        len;
    __u16               queue_mapping;
    struct net_device *dev;                 /* order of the fields doesn't matter */
};
struct {                                                    /* BTF-defined maps */
    __uint(type, BPF_MAP_TYPE_PERF_EVENT_ARRAY);
    __uint(key_size, sizeof(int));
    __uint(value_size, sizeof(int));
} perf_buf_map SEC(".maps");

struct trace_kfree_skb {
    struct sk_buff *skb;
    void *location;
};
SEC("raw_tracepoint/trace_kfree_skb")
int trace_kfree_skb(struct trace_kfree_skb* ctx)
{
    struct sk_buff *skb = ctx->skb;
    struct net_device *dev = _(skb->dev);
    int ifindex = _(dev->ifindex);

    bpf_printk("skb->len %d\n", _(skb->len));
    bpf_printk("skb->queue_mapping %d\n", _(skb->queue_mapping));
    bpf_printk("dev->ifindex %d\n", ifindex);
                          /* send first 72 bytes of the packet to user space */
    bpf_skb_output(skb, &perf_buf_map, (72ull << 32) | BPF_F_CURRENT_CPU,
                   &ifindex, sizeof(ifindex));
    return 0;
}
```

BPF verifier giant leaps in 2019

- Bounded loops
- bpf_spin_lock
- Dead code elimination
- Scalar precision tracking

BPF verifier is smarter than llvm

- The verifier removes dead code after it was optimized by llvm -O2.
- Developers cannot cheat by type casting integer to pointer or removing 'const'.

- LLVM goal -> optimize the code.
- The verifier goal -> analyze the code.

- Different takes on data flow analysis.
- The verifier data flow analysis must be precise.

BPF verifier 2.0

- The verifier cannot tell what "r2 = *(u64*)(r1 + 8)" assembly instruction is doing.
  - Unless r1 is a builtin type and +8 is checked by is_valid_access().

- The verifier cannot trust user space hints to verify BPF program assembly code.

- In-kernel BTF is trusted.

- With BTF the verifier data flow analysis enters into new realm of possibilities.

Every program type implements its own is_valid_access() and convert_ctx_access().
#1 cause for code bloat.
Bug prone code.

```
drivers/media/rc/bpf-lirc.c:    .is_valid_access = lirc_mode2_is_valid_access
kernel/bpf/cgroup.c:    .is_valid_access        = cgroup_dev_is_valid_access,
kernel/bpf/cgroup.c:    .is_valid_access        = sysctl_is_valid_access,
kernel/bpf/cgroup.c:    .is_valid_access        = cg_sockopt_is_valid_access,
kernel/trace/bpf_trace.c:       .is_valid_access = kprobe_prog_is_valid_access,
kernel/trace/bpf_trace.c:       .is_valid_access = tp_prog_is_valid_access,
kernel/trace/bpf_trace.c:       .is_valid_access = raw_tp_prog_is_valid_access,
kernel/trace/bpf_trace.c:       .is_valid_access = raw_tp_writable_prog_is_valid_access,
kernel/trace/bpf_trace.c:       .is_valid_access        = pe_prog_is_valid_access,
net/core/filter.c:      .is_valid_access        = sk_filter_is_valid_access,
net/core/filter.c:      .is_valid_access        = tc_cls_act_is_valid_access,
net/core/filter.c:      .is_valid_access        = xdp_is_valid_access,
net/core/filter.c:      .is_valid_access        = cg_skb_is_valid_access,
net/core/filter.c:      .is_valid_access        = lwt_is_valid_access,
net/core/filter.c:      .is_valid_access        = lwt_is_valid_access,
net/core/filter.c:      .is_valid_access        = lwt_is_valid_access,
net/core/filter.c:      .is_valid_access        = lwt_is_valid_access,
net/core/filter.c:      .is_valid_access        = sock_filter_is_valid_access,
net/core/filter.c:      .is_valid_access        = sock_addr_is_valid_access,
net/core/filter.c:      .is_valid_access        = sock_ops_is_valid_access,
net/core/filter.c:      .is_valid_access        = sk_skb_is_valid_access,
net/core/filter.c:      .is_valid_access        = sk_msg_is_valid_access,
net/core/filter.c:      .is_valid_access        = flow_dissector_is_valid_access,
net/core/filter.c:      .is_valid_access        = sk_reuseport_is_valid_access,
```

None of it is needed with BTF.
Will be able to remove 1000s of lines.*

* when BTF kconfig is on.

```
static u32 bpf_convert_ctx_access(enum bpf_access_type type,
                                  const struct bpf_insn *si,
                                  struct bpf_insn *insn_buf,
                                  struct bpf_prog *prog, u32 *target_size)
{
        struct bpf_insn *insn = insn_buf;
        int off;

        switch (si->off) {
        case offsetof(struct __sk_buff, len):
                *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                      bpf_target_off(struct sk_buff, len, 4,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, protocol):
                *insn++ = BPF_LDX_MEM(BPF_H, si->dst_reg, si->src_reg,
                                      bpf_target_off(struct sk_buff, protocol, 2,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, vlan_proto):
                *insn++ = BPF_LDX_MEM(BPF_H, si->dst_reg, si->src_reg,
                                      bpf_target_off(struct sk_buff, vlan_proto, 2,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, priority):
                if (type == BPF_WRITE)
                        *insn++ = BPF_STX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                              bpf_target_off(struct sk_buff, priority, 4,
                                                             target_size));
                else
                        *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                              bpf_target_off(struct sk_buff, priority, 4,
                                                             target_size));
                break;

        case offsetof(struct __sk_buff, ingress_ifindex):
                *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                      bpf_target_off(struct sk_buff, skb_iif, 4,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, ifindex):
                *insn++ = BPF_LDX_MEM(BPF_FIELD_SIZEOF(struct sk_buff, dev),
                                      si->dst_reg, si->src_reg,
                                      offsetof(struct sk_buff, dev));
                *insn++ = BPF_JMP_IMM(BPF_JEQ, si->dst_reg, 0, 1);
                *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->dst_reg,
                                      bpf_target_off(struct net_device, ifindex, 4,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, hash):
                *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                      bpf_target_off(struct sk_buff, hash, 4,
                                                     target_size));
                break;

        case offsetof(struct __sk_buff, mark):
                if (type == BPF_WRITE)
                        *insn++ = BPF_STX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                              bpf_target_off(struct sk_buff, mark, 4,
                                                             target_size));
                else
                        *insn++ = BPF_LDX_MEM(BPF_W, si->dst_reg, si->src_reg,
                                              bpf_target_off(struct sk_buff, mark, 4,
                                                             target_size));
                break;
```

How you can help

We need you

to hack.
to talk.
to invent.

BPF development is 100% use case driven.
Your requests, complains, sharing of success stories are shaping the future kernel.

JUST DO BPF.