BPF and the future of the kernel extensibility
Goal

-- This presentation focuses on past, present and the future of BPF --

• Let non-kernel developers safely and easily modify kernel behavior
  ==
• Make BPF easy to use
BPF in the past

• Either a truck or a robot
• Cool and powerful, but only in these two forms
• tcpdump, dhclient, pcap, nmap, solarflare – packet filtering
• seccomp – chrome sandbox
BPF in the present

- Giant lego set where instruction manual was not printed
BPF in the present

• Despite lack of instructions people built lots of REAL rocket ships:
  • Katran, droplet, tcpeventd, fbflow, blklatencyd, dynolog, strobelight, ttld, ila
  • Lots of BCC tools, bpftrace, ply, systemtap-bpf
  • Cilium, weaveworks, sysdig, systemd per-cgroup
• ships do look similar
Why folks learn BPF?

• NOT because it’s cool
• To solve real production issues the user space only solution is not good enough
• Kernel behavior needs to be modified
• Best solutions appear when kernel and user space work together

• when kernel is difficult to extend and roll in production, it is bypassed
  • ex: dpdk/spdk, seastar/scylladb, snabb, odp, vpp
• kernel needs BPF to stay relevant
How BPF programs look today?

- Loop-free, lock-free, short BPF programs that glue lots of kernel helpers and invoked at specified hooks
BPF hooks in tracing

- **kprobe** – read only access to arguments of any kernel function
- **uprobe** – read and write access to any user space process
- **syscalls** – read only access to syscall args
- **pmu events** (timers, hw/sw counters) – read only pt_regs
- **tracepoint** – read only access to tracepoint record defined in events/…/format
- **raw_tracepoint** – read only access to kernel internal tracepoint args
BPF hooks in networking

- sockets – read only access to skb
- XDP – raw dma buffer of the NIC
- lwt – routing in/out/xmit partial read/write of skb
- clsbpf – tc ingress/egress full read/write of skb
- cgroup scoped
  - socket create
  - L3 socket ingress/egress read only and drop
  - tcp-bpf variety (timeout_init, rwnd_init, tcp_connect, activeEstablished, passiveEstablished, needsecn, basertt, rto, retrains, state_change)
  - sockmap (L7 parsing on ingress before recvmsg with redirect) == in-kernel tcp proxy
  - device (mknod, read, write)
  - bind/connect
BPF helpers

- map access (lookup, update, delete)
- tail_call – jump into next bpf program
- perf_event_output – ring buffer communication with user space
- probe_read, probe_read_str, probe_write_user – probe kernel memory and write into user
- get_stackid – kernel/user stack collection
- ktime_get_ns, prandom, processor_id, numa_node_id
- get_current_task
- override_return – fault injection
BPF helpers in networking

- load_bytes, store_bytes – batch modify skb
- change_head, change_tail – modify skb size
- csum_replace, csum_update, csum_diff
- change_proto – ipv4->ipv6
- set/get_tunnel, push/push_vlan – encap/decap
- set_hash/get_hash
- get_socket_cookie, get_socket_uid – android traffic accounting
- setsockopt, getsockopt – tcp-bpf
- redirect – xdp and skb level redirect
- sk_redirect – L7 tcp stream redirect
BPF verifier
BPF verifier in the present

- Loop-free, lock-free, short BPF programs with single argument (context) that call BPF helpers

- BPF-to-BPF calls started new era of verifier analysis
  - arbitrary arguments (up to 5) and arbitrary return value
BPF verifier in the future

- track pointer lifetime within program (the work is done by Joe Stringer from Covalent)
  - use-case: return socket pointer from bpf helper and make sure that program does `sock_put()` on it
  - allows lock/unlock, malloc/free to be called by the program
BPF verifier in the future

- bounded loops (competing proposals from John Fastabend from Covalent and Ed Cree from Solarflare)
  - safe loops inside programs!
BPF verifier in the future

- local storage to eliminate hash lookups
- global variables
- indirect calls that are statically verified and patched
- libraries
- dynamic linking
BPF verifier in the future

- move away from existing brute force "walk all instructions" approach to proper compiler technology and static analysis
- remove #define BPF_COMPLEXITY_LIMIT 128k crutch
- remove #define BPF_MAXINSNS 4k
- support arbitrary large programs and libraries
  - 1 Million BPF instructions
- an algorithm to solve Rubik's cube will be expressible in BPF
BPF in the future

- easy to use
- easy to learn
Thank you!
Please ask questions
Part 2 is coming
Agenda

- btf update
- libbpf elf loader, elf->c codegen
- katran
- fd-based networking and cgroups
- cgroup local storage
- common driver core
- firmware no more
BTF update

- similar to old Compat Type Format yet different encoding
- landed
  - all basic types, structs, unions
  - support for map key and value
  - btf_id, btf_fd, get_next introspection
- implemented
  - support for 'function pointer type'
  - pahole -> btf
- upcoming
  - reuse 'function pointer type' to describe bpf progs
    (yet another difference vs CTF)
  - describes prog name, arguments, return type
  - integrate with verifier for safety checks
  - llvm btf backend
  - libbpf and bpftool support
- future
  - vmlinux dwarf->btf section
  - llvm pointer dereference (bpf_probe_read) annotation with symbolic field name
libbpf

- today's libbpf is elf loader only
- implemented
  - btf reader from kernel and pretty print
- future
  - write into bpf maps with btf info from kernel
  - extend with dwarf->btf converter and push btf to kernel
  - convert .o bpf files into standalone .c files where bpf progs represented as hex bytes and C code that can load progs/maps as a sequence of sys_bpf() calls.
That removes elf/dwarf from dependency for final app that compiles and links these generated .c
katran

- facebook open sourced production L4 load balancer (katran)
- https://github.com/facebookincubator/katran
- gpl-2
- key advantage enabled by XDP:
FD based APIs

- all bpf attachments points can be either global (xdp, tc, cgroup) or local (sockets, perf_events)
- global -> the progs stay attached even when user space exits
- local -> attached to FDs. auto-detach, auto-unload when user space exits
- recently added FD based kprobe, uprobe, raw_tracepoint APIs
- cgroup-bpf is difficult to get right, since cgroup can be cgroup_is_dead() or unmounted
  - centralize cgroup-bpf management into single deamon
  - introduce new cgroup-fd object just for attaching bpf to it
- convert tc ingress/egress hooks to be FD based as well
  - solves concurrency issue (multi process access to the same attach point)
  - solves autocleanup
cgroup local storage

- similar to Thread Local Storage
- new map type. one per program. value_size = requested size of local storage
- bpf_get_local_storage(map, flags)
  - cheap and fast helper to return a pointer to scratch buffer that is uniquely visible to this program only at given cgroup
- storage area allocated once at attach time of bpf prog to cgroup
- destroyed when prog is detached
- access from user space via bpf_map_lookup()/update()

- next steps
  - socket local storage and task local storage
  - clang+llvm extension:
    ```c
    __cgroup struct cgroup_buf {
      int var;
    } buf;
    int bpf_prog(ctx)
    {
      access buf.var;
    }
    ```
common driver core

- drivers are slow to add XDP support
- move memory management out of drivers into core
firmware no more

- proprietary firmware in a NIC is a huge security threat. Bigger than spectre/meltdown
- firmware used to be tiny sw shim baked into chip once
- now firmware is a monster blob full of secret features and bugs
- firmware sw teams often several times larger than driver teams
- most of the firmware logic has to become open, become part of the driver, and kernel git
- anything that can be flushed -> open
- baked in forever firmware (analog, phy, power, tpm) -> proprietary for now