Various BPF core topics.

Daniel Borkmann
<daniel@iogearbox.net>
Isovalent

lsfmm, bpf track, April, 2019

libbpf items

libbpf items

- libbpf everywhere, part 1: conversion of iproute2
  - struct bpf_elf_map's one-way compatibility
  - Fields: id, pinning, inner_id, inner_idx
  - Generalization of such extensions via BTF
    - Might need to go in line with making BTF mandatory
    - Could work given iproute2 is aligned with kernel releases
  - Getting rid of BPF_ANNOTATE_KV_PAIR()
    - Integration into struct bpf_elf_map?
  - Rest of iproute2 specifics e.g. auto-attaching tail calls less relevant for libbpf

libbpf items

- libbpf everywhere, part 2: golang bindings
  - Cilium uses iproute2 loader, also has ELF parsing in golang
  - Goal: everything out of native golang, only debugging generated object files via iproute2/bpftool
  - Both would have same behavior
  - Bindings would be under upstream under tools/lib/bpf/
  - Challenge: keeping up with libbpf pace, binding test coverage
    - Development and maintenance (us + Cloudflare + others)?
    - Proper libbpf test suite (incl. bindings) in kselftests?
    - Ideal: new features submitted along with bindings

libbpf items

- libbpf multi-object support
  - Goal: BPF-to-BPF calls and non-static global data across objects
  - BPF-to-BPF calls, what is needed?
    - Program retrieval via BTF based on function signature?
    - What about collisions, should they be rejected?
    - Would libbpf query kernel or only object based?
    - Call extension with prog fd similarly how we reference maps
    - Verifier context (input, output types) stored in prog
    - Target address for JITs via bpf_jit_get_func_addr()
  - Non-static global data, what is needed?
    - Other object's .data/.bss/.rodata map retrieval via BTF
    - Similar question as with calls: search scope and collisions?
    - Rest of workflow for kernel insertion same

BPF tail calls. Part 1: Combine with BPF-to-BPF calls

BPF tail calls. Part 1: Combine with BPF-to-BPF calls

- Heavily used inside Cilium's datapath for various reasons
  - Keeping complexity/insns down for older kernels (4.9+)
    - Some of v4, v6 handling 'outsourced' to tail calls
    - NAT46, ARP proxy, ICMP/ICMPv6 moved out of ciritical path
  - Enabling debugging on the fly (e.g. drop notifications)
  - Flexible per endpoint policy handling

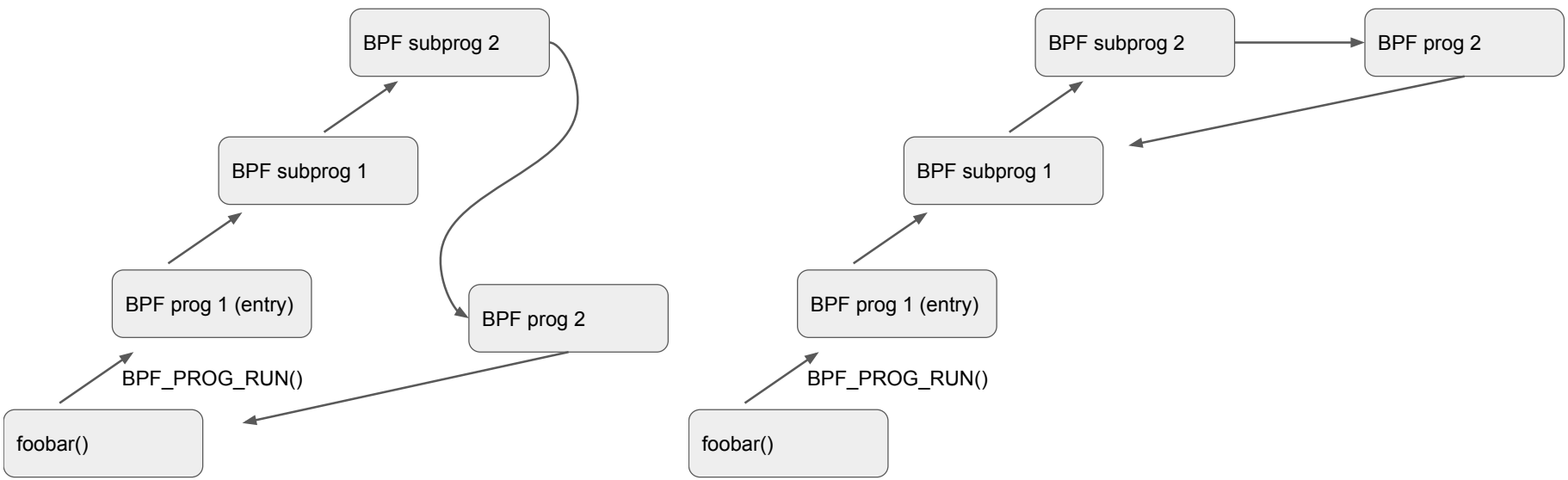
BPF tail calls. Part 1: Combine with BPF-to-BPF calls

- Problem: does not work with BPF-to-BPF calls
  - Today: everything needs __always_inline hack
  - Not very icache friendly, bloats up insn image (in some configs close to 4k already)
- Cilium programs with __always_inline removed[1]:
  - Plain replace $\rightarrow$ backend error: defined with too many args
    - BPF conntrack lookup, lb reverse NAT need rework (left inlined)
    - Solving generically: push/pop BPF insns?
  - bpf_lxc.o: shrinks prog sections up to 15%, moves 1.2k insns into .text
    - With BPF conntrack lookup expected to be $\uparrow$ %

[1] DENABLE_{IPV4,IPV6,LB_L3,LB_L4,NAT46,IPSEC,ARP_RESPONDER,LEGACY_SERVICES,MASQUERADE}

BPF tail calls. Part 1: Combine with BPF-to-BPF calls

- Possible semantics when in subprogs:
  - Tail called program replaces current program entirely
  - Tail called program replaces current sub-program only
- Both could be possible path forward (though expected might be option 1)



BPF tail calls. Part 1: Combine with BPF-to-BPF calls

- Here: tail called program replaces current program entirely
- Borrowing implicit setjmp/longjmp idea for subprogs
  - After epilogue in main program we have emitted code section for tail calls. A-priori known location for subprogs.
  - On entry we save SP at location also known to subprogs
  - From subprog where we do tail call, we save current SP, replace with main prog's one
  - Then jump to tail call section, reg1-3 content still intact
  - If tail call fails, we restore old SP and continue at next insn in subprog
  - Due to SP switch, tail call counter from main prog being used
  - Tail called prog will restore regs based on SP
- Stack: main prog (512 fixed) + subprogs ($\leq$ 512)

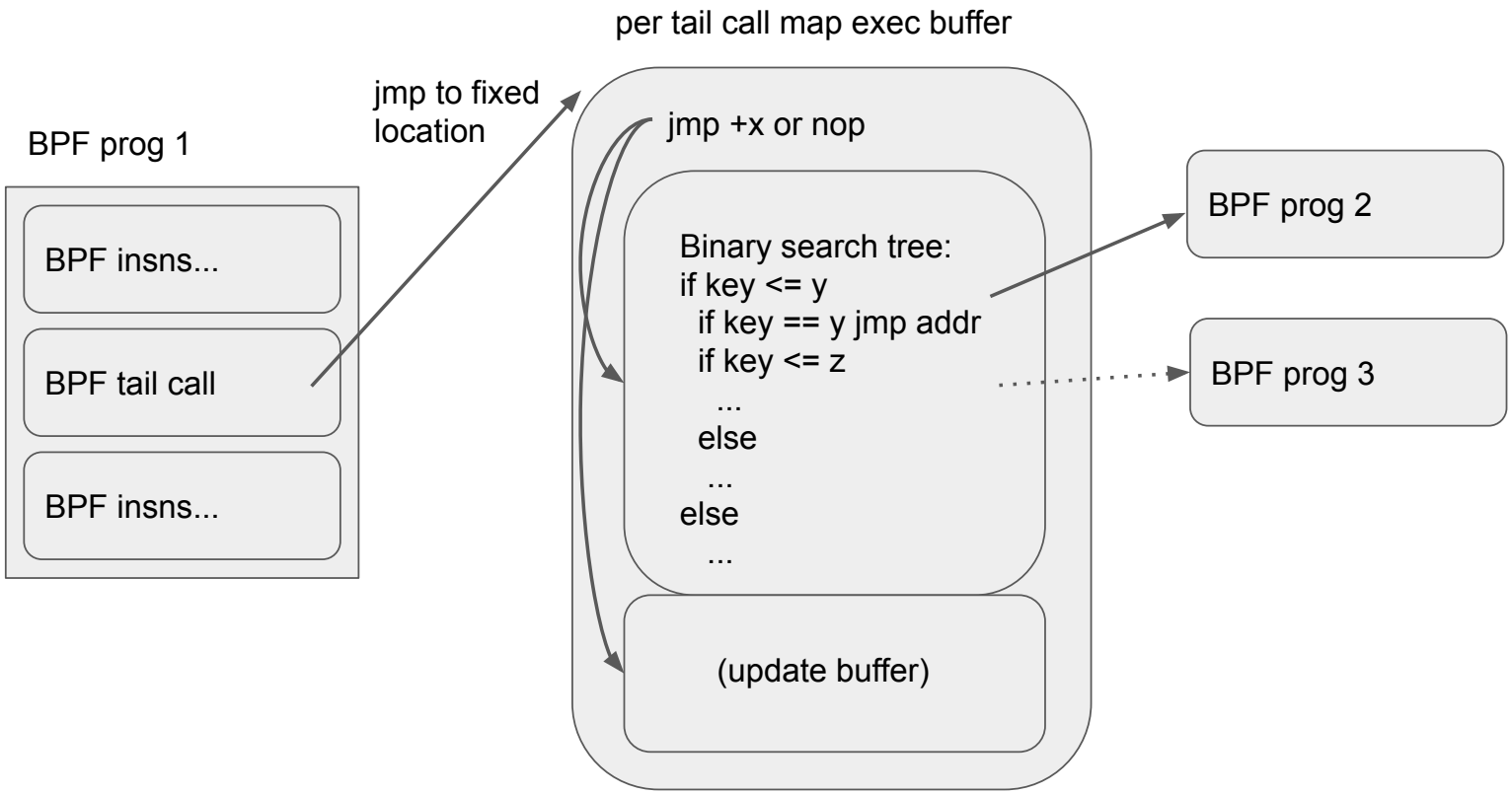
BPF tail calls. Part 1: Combine with BPF-to-BPF calls

- Here: tail called program replaces current sub-program only
  - Advantage: nothing needs to be changed for JITs
  - Verifier needs to match on expected return types
  - Types need to be enforced upon prog attach time
- Stack: main prog (512 fixed) + subprogs ($\leq$ 8 * 512)
  - How can we overcome excessive stack usage?

BPF tail calls. Part 2: Switch to direct calls

BPF tail calls. Part 2: Switch to direct calls

- Motivation: avoiding expensive retpolines
- 2 locations: BPF prog entry (e.g. XDP), tail call maps
- Tricky part: addresses can change at runtime



Misc items

Misc items

- LRU map entry eviction: flag for not marking on user space lookup
- Global data: read-mostly support $\rightarrow$ could be done as separate map
- Tooling infrastructure: barriers in installed headers