# Advanced programmability and recent updates with tc's cls_bpf.

Daniel Borkmann
<daniel@iogearbox.net>
Noiro Networks / Cisco Systems

netdev 1.2, Tokyo, October 6, 2016

netdev 1.1 talk: part 1, this talk: part 2

# Big Picture: eBPF and cls_bpf

- eBPF: efficient, generic in-kernel bytecode engine
- Today used mainly in networking, tracing, sandboxing
  - tc, XDP, socket filters/demuxing, perf, bcc, seccomp, LSM, ...
- cls_bpf programmable classifier and action in tc subsystem
- Attachable to ingress, egress of kernel's networking data path
- C → LLVM → eBPF → ELF → tc → verifier → JIT → cls_bpf → offload
- cls_bpf complementary to XDP
  - Attachable to all net devices
  - skb as input context
  - Applicable to ingress, egress

user space, kernel space

# eBPF Architecture

- 11 64bit registers, 32bit subregisters, stack, pc
- Instructions 64bit wide, max 4096 instructions/program
- Various new instructions over cBPF
- Core components of architecture
  - Read/write access to context
  - Helper function concept
  - Maps, arbitrary sharing
  - Tail calls
  - Object pinning
  - cBPF to eBPF translator
  - LLVM eBPF backend
- eBPF JIT backends implemented by archs
- Management via `bpf(2)`, stable ABI

# cls_bpf and sch_clsact

- sch_clsact container for tc classifier and actions
- Provides two central hooks in data path
  - Ingress: __netif_receive_skb_core()
  - Egress: __dev_queue_xmit()
- cls_bpf runs eBPF, allows for atomic updates
- Fast-path with direct-action (da) mode
  - Verdicts: ok, shot, stolen, redirect, unspec
- Offload interface implementable by drivers
- tc eBPF frontend as ELF loader
  - Parsing of sections
  - Relocation handling
  - Object pinning/retrieving

# Usage Example: Setup and Teardown

```
(Example code: see paper, kernel/iproute2 samples)
$ clang -O2 -target bpf -o foo.o -c foo.c

# tc qdisc  add dev em1 clsact
# tc qdisc show dev em1
[...]
qdisc clsact ffff: parent ffff:fff1

# tc filter add dev em1 ingress bpf da obj foo.o sec p1
# tc filter add dev em1 egress  bpf da obj foo.o sec p2

# tc filter show dev em1 ingress   (or egress)
filter protocol all pref 49152 bpf
filter protocol all pref 49152 bpf handle 0x1 foo.o:[p1] direct-action

# tc filter del dev em1 ingress
# tc filter del dev em1 egress

# tc qdisc  del dev em1 clsact
```

# Tunneling and Encapsulation

- Scalable support through collect metadata interface
  - vxlan, geneve, gre, ipip, ipip6, ip6ip6
- Key is translated from BPF representation into tunnel info
  - id, v4/v6 dst ip, tos, ttl, label, flags (csum, proto, frag)
- Option is passed as raw blob
  - vxlan gbp, geneve TLVs
- RX via `struct metadata_dst` from `skb`
- TX as per-CPU `struct metadata_dst` temporarily set to `skb`
- eBPF helpers
  - `bpf_skb_{get,set}_tunnel_key()`
  - `bpf_skb_{get,set}_tunnel_opt()`

# Direct Packet Access

- Available methods prior to direct packet access
  - `BPF_LD|BPF_ABS` and `BPF_LD|BPF_IND`
    - Carried over from cBPF
    - LLVM built-in helper: `asm("llvm.bpf.load.byte"), ...`
    - 1, 2, 4 byte load into register
    - Host endianess
    - Suboptimal exception handling
    - Fast path implemented by JITs
    - Slow path call for non-linear data, negative offsets
  - `bpf_skb_load_bytes()`
    - Helper wrapper for `skb_header_pointer()`
    - Therefore no JIT/LLVM/endianess special handling
    - 1-X byte load into stack space
    - Limited by eBPF stack space itself
    - Exception handling possible

# Direct Packet Access

- Available methods prior to direct packet access
  - `BPF_LD|BPF_ABS` and `BPF_LD|BPF_IND`
    - Carried over from cBPF
    - LLVM built-in helper: `asm("llvm.bpf.load.byte"), ...`
    - 1, 2, 4 byte load into register
    - Host endianess
    - Suboptimal exception handling
    - Fast path implemented by JITs
    - Slow path call for non-linear data, negative offsets
  - `bpf_skb_load_bytes()`
    - Helper wrapper for `skb_header_pointer()`
    - Therefore no JIT/LLVM/endianess special handling
    - 1-X byte load into stack space
    - Limited by eBPF stack space itself
    - Exception handling possible

# Direct Packet Access

- Available methods prior to direct packet access
  - `bpf_skb_store_bytes()`
    - Helper call, thus same properties as `bpf_skb_load_bytes()`
    - Unclones skb, pulls in non-linear data if needed
    - Flags for csum update, clearing hash
- Direct packet access
  - Combining advantages of both
    - New `data`, `data_end` members for skb context
    - Loaded into register, access `skb→data` directly
    - No JIT/LLVM special handling needed
    - Complexity rather pushed into verifier, not runtime
    - Matches on `data + X` vs. `data_end` test, tracks ranges
    - Implicit exception handling from branches
    - Write part strictly uncloned, helper for non-linear data

# Direct Packet Access

- Available methods prior to direct packet access
  - `bpf_skb_store_bytes()`
    - Helper call, thus same properties as `bpf_skb_load_bytes()`
    - Unclones skb, pulls in non-linear data if needed
    - Flags for csum update, clearing hash
- Direct packet access
  - Combining advantages of both
    - New `data`, `data_end` members for skb context
    - Loaded into register, access `skb`→`data` directly
    - No JIT/LLVM special handling needed
    - Complexity rather pushed into verifier, not runtime
    - Matches on `data + X` vs. `data_end` test, tracks ranges
    - Implicit exception handling from branches
    - Write part strictly uncloned, helper for non-linear data

# Event Output/Notifications

- Idea: event push mechanism from kernel → user space direction
- Per-cpu lockless `mmap(2)` ring buffer from perf infrastructure
- Busy-poll or possible wake-up defineable for #events, #bytes
- Ring buffer slot layout fully programmable, not part of uapi
- Use-cases: sampling, monitoring, debugging, management daemons
- Used in `cilium` project as
  - Drop monitor for policy learning
  - Packet tracing infrastructure
  - `bpf_trace_printk()` replacement

# JITs, Offload, Hardening

- Available as of today: `x86_64`, `arm64`, `ppc64`, `s390`
- `ppc64`: initial JIT merged and tail call support added
- `arm64`: tail call support, various optimizations, xadd still missing
- Offloading of `cls_bpf` with eBPF to NIC
  - Supported by Netronome SmartNICs via JIT (Jakub's, Nic's talk[1])
- Various hardening measures done by default (RO, rand gap)
- Constant blinding infrastructure: `net.core.bpf_jit_harden=1`
  - Blinding for non-root programs enabled
  - Rewriting 32/64bit constants generically at BPF instruction level
  - $\texttt{imm} \rightarrow ((\texttt{rnd} \oplus \texttt{imm}) \oplus \texttt{rnd})$, $\texttt{ins}_{\text{imm}} \rightarrow \texttt{ins}_{\text{reg}}$

[1] "eBPF/XDP hardware offload to SmartNICs", netdev 1.2

# Constant Blinding

- x86_64 JIT example for `BPF_LD|BPF_IMM`:

```
b8 XX YY ZZ a8         mov $0xa8ZZYYXX, %eax
b8 PP QQ RR a8         mov $0xa8RRQQPP, %eax
b8 ...
```

- Off-by-one jump ...

```
XX YY ZZ               payload insn
a8 b8                  test $0xb8, %al
PP QQ RR               payload insn
a8 b8                  test $0xb8, %al
...
```

- Blinded, `mov` case rewritten as `mov/xor/mov`, e.g.

```
41 ba 63 25 19 e1      mov $0xe1192563,%r10d
41 81 f2 f3 b5 89 49   xor $0x4989b5f3,%r10d
44 89 d0               mov %r10d,%eax
...
```

# Summary on Functionality

- __sk_buff context as mapper for skb metadata access
- Various helpers available for cls_bpf, main areas:
  - Packet access and mangling
  - Map (e.g. per cpu, prealloced) access
  - Checksum mangling
  - Redirection/forwarding
  - Cgroups v1/v2 integration
  - Encapsulations
  - Protocol migration (v4/v6)
  - Packet size mangling
  - Event output, debugging
  - Routing realms
  - Tail call invocation
  - Misc things (hash, cpu, random, ktime, etc)

# Thanks!

- Couple of next steps
  - Collect metadata-like API for crypto integration
  - Verifier logging improvements, code annotations
  - Better introspection facilities, code signing, etc
  - Integration into kernel selftesting framework
  - Get documentation closer to implementation status
- Code
  - `git.kernel.org` → kernel, iproute2 tree
  - cilium project: `github.com/cilium`
    - BPF & XDP for containers
- Further information
  - netdev1.1, netdev1.2 paper on `cls_bpf`
  - Kernel tree: `Documentation/networking/filter.txt`
  - Man pages: `bpf(2)`, `tc-bpf(7)`